

# Introduction to Discrete Structures

Prepared by

*Prof. | Wael Abd Elkader Awad*

Professor of Computer Science

Faculty of Computers and Artificial Intelligence

Damietta University

## Preface:

### 1. Course Syllabus:

<b>BS102</b>	<b>Discrete Structures</b>	تراتيب محددة
--------------	----------------------------	--------------

Graph, lattices, Trees; Algebraic Structures: semi-group, group, integer congruence's; asymptotic notation and growth of functions; permutations and combinations, counting principles; Recursive definition; state machines and invariants; recurrences; generating functions; Modeling Arithmetic, Computation, and Languages.

### Prerequisites:

### 2. Exam Degree

#### مادة (17) نظام الامتحانات:

أ- يتم تصحيح امتحان كل مقرر من 100 درجة.

ب- الحد الأدنى للنجاح في المقرر الدراسي هو 50% من الدرجة النهائية.

ج- توزع درجات الامتحان في كل مقرر على النحو التالي:

المقرر نظري أو له تمارين	المقرر نظري وعملي	نوع الامتحان
20	15	امتحان نصف فصلی نظري
10	10	امتحان شفوي نهائي
-	15	امتحان عملي نهائي
10	10	تقييم مستمر - تكليفات
60	50	امتحان نظري نهائي
100	100	مجموع درجات المقرر

### 3. Evaluation System

#### مادة (18) نظام التقييم:

أ- تتبع الكلية نظام الساعات المعتمدة والذي يعتمد على أن الوحدة الأساسية هي الساعة المعتمدة، ويكون نظام التقييم على أساس النتائج في كل مقرر دراسي بنظام النقاط والذي يحدد طبقاً للجدول التالي:

النقطة	التقدير	النسبة المئوية للدرجة
4	A+	%96 فأكثر
3.7	A	%92 - أقل من %96
3.4	A-	%88 - أقل من %92
3.2	B+	%84 - أقل من %88
3	B	%80 - أقل من %84
2.8	B-	%76 - أقل من %80
2.6	C+	%72 - أقل من %76
2.4	C	%68 - أقل من %72
2.2	C-	%64 - أقل من %68
2	D+	%60 - أقل من %64
1.5	D	%55 - أقل من %60
1	D-	%50 - أقل من %55
صفر	F	أقل من %50
صفر	Abs	غياب عن حضور الامتحان النهائي بدون عذر مقبول من مجلس الكلية
بدون نقاط مع عدم احتساب عدد الساعات ضمن المعدل التراكمي إلا بعد الانتهاء من دراسة المقرر سواء بالنجاح أو الرسوب		مقرر مستمر في الفصل التالي
		I مقرر غير مكتمل
		W الانسحاب من مقرر

### 4. Degree Distribution

الدرجة	البيان	م
20	امتحان نصف فصلٍ نظري	1
10	تقييم مستمر - تكليفات	2
10	امتحان شفويٌ نهائيٌ	3
60	امتحان نهاية الفصل الدراسي	4
100	المجموع	

### 5. Textbook

- Textbook:

B. Kolman, R.C. Busby & S.C. Ross,  
Discrete Mathematical Structures (Sixth Edition), Higher Education Press, 2010.11.

# Table of Contents

Subject	Page No.
<b>Chapter (1): Sets, Functions, and Induction</b>	
Preface	
1.1 Sets	1
1.2 Relations	5
1.3 Functions	7
1.4 Set Cardinality, revisited	8
2.1 Basic Proof Techniques	13
2.2 Proof by Cases and Examples	15
2.3 Induction	17
2.4 Inductive Definitions	26
Exercises	37
<b>Chapter (2): Propositional Logic</b>	
1.1 Propositional Logic	39
1.2 Propositional Equivalences	54
1.3 Rules of Inference	66
Exercises	72
<b>Chapter (3): Recursion and Counting</b>	
1.1 The Basics of Counting	75
1.2 Permutations and Combinations	96
1.3 The Pigeonhole Principle	104
Exercises	112
<b>Chapter (4): Graph Theory</b>	
1.1 Introduction of Graphs	116
1.2 Types of Graphs	118
1.3 Special Types of Graphs	142
1.4 Graph Operations	150
1.5 Paths and cycles	160
1.6 Representing Graphs on Computers	175
Exercises	193

Subject	Page No.
<b>Chapter (5): Trees</b>	
1.1 Introduction to Trees	201
1.2 The terminology for trees	206
1.3 Properties of Trees	211
1.4 Balanced binary trees	219
1.5 Full binary tree	223
1.6 Complete binary tree	225
1.7 Binary Search Trees	227
1.8 Spanning Trees	241
Exercises	258
<b>Chapter (6): Algorithms, the Growth of Functions</b>	
1.1 Analysis of Algorithms	264
1.2 Types of Analysis	270
1.3 Asymptotic Analysis	274
1.4 Complexity classes	287
1.5 Properties of Asymptotic Notations	304
Exercises	312
<b>Chapter (7): Recursive and Generating Functions</b>	
1.1 Sequences	320
1.2 Closed formula	322
1.3 Recursive Definitions	324
1.4 Geometric Sequences	334
1.5 Generating Functions	340
Exercises	355
<b>Chapter (8): Modeling Computations</b>	
1.1 Languages and Grammars	362
1.2 Types of Phrase-Structure Grammars	366
1.3 Finite-State Machines with Output	373
1.4 Finite-State Machines with No Output	380
1.5 Language Recognition	393
1.6 Turing Machines	403
Exercises	412
<b>Bibliography</b>	<b>420</b>

# **Chapter (1)**

## **Sets, Functions, and Induction**

## Chapter 1

# Sets, Functions and Relations

*“A happy person is not a person in a certain set of circumstances, but rather a person with a certain set of attitudes.”*  
– Hugh Downs

### 1.1 Sets

A set is one of the most fundamental object in mathematics.

**Definition 1.1** (Set, informal). A set is an *unordered* collections of objects.

Our definition is informal because we do not define what a “collection” is; a deeper study of sets is out of the scope of this course.

**Example 1.2.** The following notations all refer to the same set:

$$\{1, 2\}, \{2, 1\}, \{1, 2, 1, 2\}, \{x \mid x \text{ is an integer, } 1 \leq x \leq 2\}$$

The last example read as “the set of all  $x$  such that  $x$  is an integer between 1 and 2 (inclusive)”.

We will encounter the following sets and notations throughout the course:

- $\emptyset = \{\}$ , the empty set.
- $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ , the non-negative integers
- $\mathbb{N}^+ = \{1, 2, 3, \dots\}$ , the positive integers
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ , the integers
- $\mathbb{Q} = \{q \mid q = a/b, a, b \in \mathbb{Z}, b \neq 0\}$ , the rational numbers
- $\mathbb{Q}^+ = \{q \mid q \in \mathbb{Q}, q > 0\}$ , the positive rationals
- $\mathbb{R}$ , the real numbers

- $\mathbb{R}^+$ , the positive reals

Given a collection of objects (a set), we may want to know how large is the collection:

**Definition 1.3** (Set cardinality). The cardinality of a set  $A$  is the number of (distinct) objects in  $A$ , written as  $|A|$ . When  $|A| \in \mathbb{N}$  (a finite integer),  $A$  is a finite set; otherwise  $A$  is an infinite set. We discuss the cardinality of infinite sets later.

**Example 1.4.**  $|\{1, 2, 3\}| = |\{\{1, 2\}, \{1, 2\}\}| = 3$ .

Given two collections of objects (two sets), we may want to know if they are equal, or if one collection contains the other. These notions are formalized as set equality and subsets:

**Definition 1.5** (Set equality). Two sets  $S$  and  $T$  are equal, written as  $S = T$ , if  $S$  and  $T$  contains exactly the same elements, i.e., for every  $x$ ,  $x \in S \leftrightarrow x \in T$ .

**Definition 1.6** (Subsets). A set  $S$  is a subset of set  $T$ , written as  $S \subseteq T$ , if every element in  $S$  is also in  $T$ , i.e., for every  $x$ ,  $x \in S \rightarrow x \in T$ . Set  $S$  is a strict subset of  $T$ , written as  $S \subset T$  if  $S \subseteq T$ , and there exist some element  $x \in T$  such that  $x \notin S$ .

**Example 1.7.**

- $\{1, 2\} \subseteq \{1, 2, 3\}$ .
- $\{1, 2\} \subset \{1, 2, 3\}$ .
- $\{1, 2, 3\} \subseteq \{1, 2, 3\}$ .
- $\{1, 2, 3\} \not\subseteq \{1, 2, 3\}$ .
- For any set  $S$ ,  $\emptyset \subseteq S$ .
- For every set  $S \neq \emptyset$ ,  $\emptyset \subset S$ .
- $S \subseteq T$  and  $T \subseteq S$  if and only if  $S = T$ .

Finally, it is time to formalize operations on sets. Given two collection of objects, we may want to merge the collections (set union), identify the objects in common (set intersection), or identify the objects unique to one collection (set difference). We may also be interested in knowing all possible ways of picking one object from each collection (Cartesian product), or all possible ways of picking some objects from just one of the collections (power set).

**Definition 1.8** (Set operations). Given sets  $S$  and  $T$ , we define the following operations:

- *Power Sets.*  $\mathcal{P}(S)$  is the set of all subsets of  $S$ .
- *Cartesian Product.*  $S \times T = \{(s, t) \mid s \in S, t \in T\}$ .
- *Union.*  $S \cup T = \{x \mid x \in S \text{ or } x \in T\}$ , set of elements in  $S$  or  $T$ .
- *Intersection.*  $S \cap T = \{x \mid x \in S, x \in T\}$ , set of elements in  $S$  and  $T$ .
- *Difference.*  $S - T = \{x \mid x \in S, x \notin T\}$ , set of elements in  $S$  but not  $T$ .
- *Complements.*  $\overline{S} = \{x \mid x \notin S\}$ , set of elements not in  $S$ . This is only meaningful when we have an implicit universe  $\mathcal{U}$  of objects, i.e.,  $\overline{S} = \{x \mid x \in \mathcal{U}, x \notin S\}$ .

**Example 1.9.** Let  $S = \{1, 2, 3\}$ ,  $T = \{3, 4\}$ ,  $V = \{a, b\}$ . Then:

- $\mathcal{P}(T) = \{\emptyset, \{3\}, \{4\}, \{3, 4\}\}$ .
- $S \times V = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$ .
- $S \cup T = \{1, 2, 3, 4\}$ .
- $S \cap T = \{3\}$ .
- $S - T = \{1, 2\}$ .
- If we are dealing with the set of all integers,  $\overline{S} = \{\dots, -2, -1, 0, 4, 5, \dots\}$ .

Some set operations can be visualized using Venn diagrams. See Figure 1.1. To give an example of working with these set operations, consider the following set identity.

**Theorem 1.10.** For all sets  $S$  and  $T$ ,  $S = (S \cap T) \cup (S - T)$ .

*Proof.* We can visualize the set identity using Venn diagrams (see Figure 1.1b and 1.1c). To formally prove the identity, we will show both of the following:

$$S \subseteq (S \cap T) \cup (S - T) \quad (1.1)$$

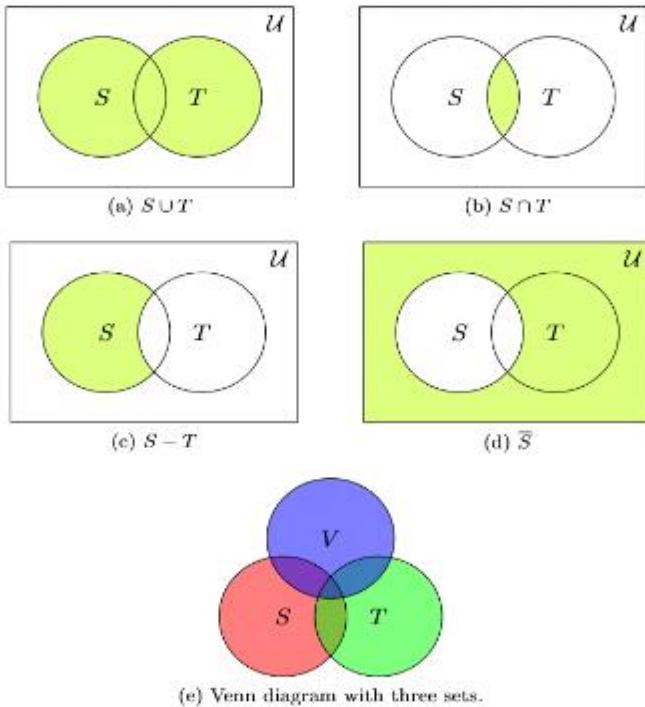
$$(S \cap T) \cup (S - T) \subseteq S \quad (1.2)$$

To prove (1.1), consider any element  $x \in S$ . Either  $x \in T$  or  $x \notin T$ .

- If  $x \in T$ , then  $x \in S \cap T$ , and thus also  $x \in (S \cap T) \cup (S - T)$ .
- If  $x \notin T$ , then  $x \in (S - T)$ , and thus again  $x \in (S \cap T) \cup (S - T)$ .

To prove (1.2), consider any  $x \in (S \cap T) \cup (S - T)$ . Either  $x \in S \cap T$  or  $x \in S - T$

- If  $x \in S \cap T$ , then  $x \in S$

Figure 1.1: Venn diagrams of sets  $S$ ,  $T$ , and  $V$  under universe  $\mathcal{U}$ .

- If  $x \in S - T$ , then  $x \in S$ . ■

In computer science, we frequently use the following additional notation (these notation can be viewed as short hands):

**Definition 1.11.** Given a set  $S$  and a natural number  $n \in \mathbb{N}$ ,

- $S^n$  is the set of length  $n$  “strings” (equivalently  $n$ -tuples) with alphabet  $S$ . Formally we define it as the product of  $n$  copies of  $S$  (i.e.,  $S \times S \times \dots \times S$ ).
- $S^*$  is the set of finite length “strings” with alphabet  $S$ . Formally we define it as the union of  $S^0 \cup S^1 \cup S^2 \cup \dots$ , where  $S^0$  is a set that contains only one element: the empty string (or the empty tuple “()”).

- $[n]$  is the set  $\{0, 1, \dots, n - 1\}$ .

Commonly seen set includes  $\{0, 1\}^n$  as the set of  $n$ -bit strings, and  $\{0, 1\}^*$  as the set of finite length bit strings. Also observe that  $|[n]| = n$ .

Before we end this section, let us revisit our informal definition of sets: an unordered “collection” of objects. In 1901, Russel came up with the following “set”, known as Russel’s paradox<sup>1</sup>:

$$S = \{x \mid x \notin x\}$$

That is,  $S$  is the set of all sets that don’t contain themselves as an element. This might seem like a natural “collection”, but is  $S \in S$ ? It’s not hard to see that  $S \in S \leftrightarrow S \notin S$ . The conclusion today is that  $S$  is not a good “collection” of objects; it is not a set.

So how will know if  $\{x \mid x \text{ satisfies some condition}\}$  is a set? Formally, sets can be defined axiomatically, where only collections constructed from a careful list of rules are considered sets. This is outside the scope of this course. We will take a short cut, and restrict our attention to a well-behaved universe. Let  $E$  be all the objects that we are interested in (numbers, letters, etc.), and let  $\mathcal{U} = E \cup \mathcal{P}(E) \cup \mathcal{P}(\mathcal{P}(E))$ , i.e.,  $E$ , subsets of  $E$  and subsets of subsets of  $E$ . In fact, we may extend  $\mathcal{U}$  with three power set operations, or indeed any *finite* number of power set operations. Then,  $S = \{x \mid x \in \mathcal{U} \text{ and some condition holds}\}$  is always a set.

## 1.2 Relations

**Definition 1.12** (Relations). A relation on sets  $S$  and  $T$  is a subset of  $S \times T$ . A relation on a single set  $S$  is a subset of  $S \times S$ .

**Example 1.13.** “Taller-than” is a relation on people;  $(A, B) \in$  “Taller-than” if person  $A$  is taller than person  $B$ . “ $\geq$ ” is a relation on  $\mathbb{R}$ ; “ $\geq$ ” =  $\{(x, y) \mid x, y \in \mathbb{R}, x \geq y\}$ .

**Definition 1.14** (Reflexivity, symmetry, and transitivity). A relation  $R$  on set  $S$  is:

- *Reflexive* if  $(x, x) \in R$  for all  $x \in S$ .
- *Symmetric* if whenever  $(x, y) \in R$ ,  $(y, x) \in R$ .

---

<sup>1</sup>A folklore version of this paradox concerns itself with barbers. Suppose in a town, the only barber shaves all and only those men in town who do not shave themselves. This seems perfectly reasonable, until we ask: Does the barber shave himself?

- *Transitive* if whenever  $(x, y), (y, z) \in R$ , then  $(x, z) \in R$

**Example 1.15.**

- “ $\leq$ ” is reflexive, but “ $<$ ” is not.
- “sibling-of” is symmetric, but “ $\leq$ ” and “sister-of” is not.
- “sibling-of”, “ $\leq$ ”, and “ $<$ ” are all transitive, but “parent-of” is not (“ancestor-of” is transitive, however).

**Definition 1.16** (Graph of relations). The graph of a relation  $R$  over  $S$  is an directed graph with nodes corresponding to elements of  $S$ . There is an edge from node  $x$  to  $y$  if and only if  $(x, y) \in R$ . See Figure 1.2.

**Theorem 1.17.** Let  $R$  be a relation over  $S$ .

- *R is reflexive iff its graph has a self-loop on every node.*
- *R is symmetric iff in its graph, every edge goes both ways.*
- *R is transitive iff in its graph, for any three nodes x, y and z such that there is an edge from x to y and from y to z, there exist an edge from x to z.*
- *More naturally, R is transitive iff in its graph, whenever there is a path from node x to node y, there is also a direct edge from x to y.*

*Proof.* The proofs of the first three parts follow directly from the definitions. The proof of the last bullet relies on induction; we will revisit it later. ■

**Definition 1.18** (Transitive closure). The transitive closure of a relation  $R$  is the least (i.e., smallest) transitive relation  $R^*$  such that  $R \subseteq R^*$ .

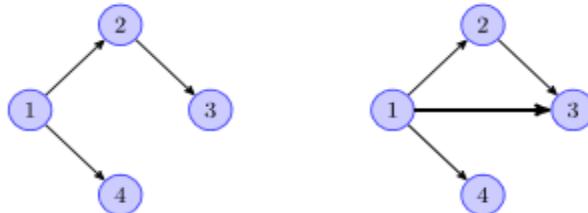
Pictorially,  $R^*$  is the connectivity relation: if there is a path from  $x$  to  $y$  in the graph of  $R$ , then  $(x, y) \in R^*$ .

**Example 1.19.** Let  $R = \{(1, 2), (2, 3), (1, 4)\}$  be a relation (say on set  $\mathbb{Z}$ ). Then  $(1, 3) \in R^*$  (since  $(1, 2), (2, 3) \in R$ ), but  $(2, 4) \notin R^*$ . See Figure 1.2.

**Theorem 1.20.** A relation  $R$  is transitive iff  $R = R^*$ .

**Definition 1.21** (Equivalence relations). A relation  $R$  on set  $S$  is an equivalence relation if it is reflexive, symmetric and transitive.

Equivalence relations capture the every day notion of “being the same” or “equal”.



(a) The relation  $R = \{(1, 2), (2, 3), (1, 4)\}$  (b) The relation  $R^*$ , transitive closure of  $R$

Figure 1.2: The graph of a relation and its transitive closure.

**Example 1.22.** The following are equivalence relations:

- Equality, “ $=$ ”, a relation on numbers (say  $\mathbb{N}$  or  $\mathbb{R}$ ).
- Parity  $= \{(x, y) \mid x, y \text{ are both even or both odd}\}$ , a relation on integers.

### 1.3 Functions

**Definition 1.23.** A function  $f : S \rightarrow T$  is a “mapping” from elements in set  $S$  to elements in set  $T$ . Formally,  $f$  is a relation on  $S$  and  $T$  such that for each  $s \in S$ , there exists a unique  $t \in T$  such that  $(s, t) \in R$ .  $S$  is the *domain* of  $f$ , and  $T$  is the *range* of  $f$ .  $\{y \mid y = f(x) \text{ for some } x \in S\}$  is the *image* of  $f$ .

We often think of a function as being characterized by an algebraic formula, e.g.,  $y = 3x - 2$  characterizes the function  $f(x) = 3x - 2$ . Not all formulas characterize a function, e.g.  $x^2 + y^2 = 1$  is a relation (a circle) that is not a function (no unique  $y$  for each  $x$ ). Some functions are also not easily characterized by an algebraic expression, e.g., the function mapping past dates to recorded weather.

**Definition 1.24 (Injection).**  $f : S \rightarrow T$  is injective (one-to-one) if for every  $t \in T$ , there exists at most one  $s \in S$  such that  $f(s) = t$ . Equivalently,  $f$  is injective if whenever  $s \neq s$ , we have  $f(s) \neq f(s)$ .

**Example 1.25.**

- $f : \mathbb{N} \rightarrow \mathbb{N}$ ,  $f(x) = 2x$  is injective.

- $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ ,  $f(x) = x^2$  is injective.
- $f : \mathbb{R} \rightarrow \mathbb{R}$ ,  $f(x) = x^2$  is not injective since  $(-x)^2 = x^2$ .

**Definition 1.26** (Surjection).  $f : S \rightarrow T$  is surjective (onto) if the image of  $f$  equals its range. Equivalently, for every  $t \in T$ , there exists some  $s \in S$  such that  $f(s) = t$ .

**Example 1.27.**

- $f : \mathbb{N} \rightarrow \mathbb{N}$ ,  $f(x) = 2x$  is not surjective.
- $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ ,  $f(x) = x^2$  is surjective.
- $f : \mathbb{R} \rightarrow \mathbb{R}$ ,  $f(x) = x^2$  is not injective since negative reals don't have real square roots.

**Definition 1.28** (Bijection).  $f : S \rightarrow T$  is bijective, or a *one-to-one correspondence*, if it is injective and surjective.

See Figure 1.3 for an illustration of injections, surjections, and bijections.

**Definition 1.29** (Inverse relation). Given a function  $f : S \rightarrow T$ , the inverse relation  $f^{-1}$  on  $T$  and  $S$  is defined by  $(t, s) \in f^{-1}$  if and only if  $f(s) = t$ .

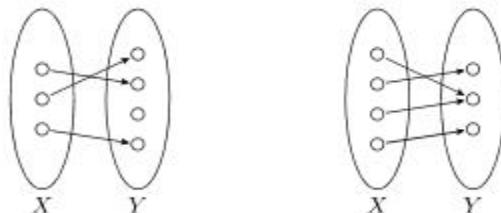
If  $f$  is bijective, then  $f^{-1}$  is a function (unique inverse for each  $t$ ). Similarly, if  $f$  is injective, then  $f^{-1}$  is also a function if we restrict the domain of  $f^{-1}$  to be the image of  $f$ . Often an easy way to show that a function is one-to-one is to exhibit such an inverse mapping. In both these cases,  $f^{-1}(f(x)) = x$ .

## 1.4 Set Cardinality, revisited

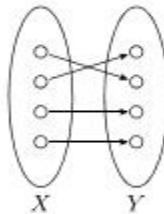
Bijections are very useful for showing that two sets have the same number of elements. If  $f : S \rightarrow T$  is a bijection and  $S$  and  $T$  are finite sets, then  $|S| = |T|$ . In fact, we will extend this definition to infinite sets as well.

**Definition 1.30** (Set cardinality). Let  $S$  and  $T$  be two potentially infinite sets.  $S$  and  $T$  have the same cardinality, written as  $|S| = |T|$ , if there exists a bijection  $f : S \rightarrow T$  (equivalently, if there exists a bijection  $f' : T \rightarrow S$ ).  $T$  has cardinality at larger or equal to  $S$ , written as  $|S| \leq |T|$ , if there exists an injection  $g : S \rightarrow T$  (equivalently, if there exists a surjection  $g' : T \rightarrow S$ ).

To “intuitively justify” Definition 1.30, see Figure 1.3. The next theorem shows that this definition of cardinality corresponds well with our intuition for size: if both sets are at least as large as the other, then they have the same cardinality.



(a) An injective function from  $X$  to  $Y$ .  
 (b) A surjective function from  $X$  to  $Y$ .



(c) A bijective function from  $X$  to  $Y$ .

Figure 1.3: Injective, surjective and bijective functions.

**Theorem 1.31** (Cantor-Bernstein-Schroeder). *If  $|S| \leq |T|$  and  $|T| \leq |S|$ , then  $|S| = |T|$ . In other words, given injective maps,  $g : S \rightarrow T$  and  $h : T \rightarrow S$ , we can construct a bijection  $f : S \rightarrow T$ .*

We omit the proof of Theorem 1.31; interested readers can easily find multiple flavours of proofs online. Set cardinality is much more interesting when the sets are infinite. The cardinality of the natural numbers is extra special, since you can “count” the numbers. (It is also the “smallest infinite set”, a notion that is outside the scope of this course.)

**Definition 1.32.** A set  $S$  is countable if it is finite or has the same cardinality as  $\mathbb{N}^+$ . Equivalently,  $S$  is countable if  $|S| \leq |\mathbb{N}^+|$ .

**Example 1.33.**

- $\{1, 2, 3\}$  is countable because it is finite.
- $\mathbb{N}$  is countable because it has the same cardinality as  $\mathbb{N}^+$ ; consider  $f : \mathbb{N}^+ \rightarrow \mathbb{N}$ ,  $f(x) = x - 1$ .

- The set of positive even numbers,  $S = \{2, 4, \dots\}$ , is countable consider  $f : \mathbb{N}^+ \rightarrow S$ ,  $f(x) = 2x$ .

**Theorem 1.34.** *The set of positive rational numbers  $\mathbb{Q}^+$  are countable.*

*Proof.*  $\mathbb{Q}^+$  is clearly not finite, so we need a way to count  $\mathbb{Q}^+$ . Note that double counting, triple counting, even counting some element infinite many times is okay, as long as we eventually count *all* of  $\mathbb{Q}^+$ . I.e., we implicitly construct a surjection  $f : \mathbb{N}^+ \rightarrow \mathbb{Q}^+$ .

Let us count in the following way. We first order the rational numbers  $p/q$  by the value of  $p + q$ ; then we break ties by ordering according to  $p$ . The ordering then looks like this:

- First group ( $p + q = 2$ ):  $1/1$
- Second group ( $p + q = 3$ ):  $1/2, 2/1$
- Third group ( $p + q = 4$ ):  $1/3, 2/2, 3/1$

Implicitly, we have  $f(1) = 1/1$ ,  $f(2) = 1/2$ ,  $f(3) = 2/1$ , etc. Clearly,  $f$  is a surjection. See Figure 1.4 for an illustration of  $f$ . ■

1/1	1/2	1/3	1/4	1/5	...
2/1	2/2	2/3	2/4	2/5	...
3/1	3/2	3/3	3/4	3/5	...
4/1	4/2	4/3	4/4	4/5	...
5/1	5/2	5/3	5/4	5/5	...
:	:	:	:	:	

Figure 1.4: An infinite table containing all positive rational numbers (with repetition). The red arrow represents how  $f$  traverses this table—how we count the rationals.

**Theorem 1.35.** *There exists sets that are not countable.*

*Proof.* Here we use Cantor's diagonalization argument. Let  $S$  be the set of infinite sequences  $(d_1, d_2, \dots)$  over digits  $\{0, 1\}$ . Clearly  $S$  is infinite. To

show that there cannot be a bijection with  $\mathbb{N}^+$ , we proceed by contradiction. Suppose  $f : \mathbb{N}^+ \rightarrow S$  is a bijection. We can then enumerate these strings using  $f$ , producing a 2-dimensional table of digits:

$$\begin{aligned} f(1) &= s^1 = (d_1^1, d_2^1, d_3^1, \dots) \\ f(2) &= s^2 = (d_1^2, d_2^2, d_3^2, \dots) \\ f(3) &= s^3 = (d_1^3, d_2^3, d_3^3, \dots) \end{aligned}$$

Now consider  $s^* = (1 - d_1^1, 1 - d_2^2, 1 - d_3^3, \dots)$ , i.e., we are taking the diagonal of the above table, and flipping all the digits. Then for any  $n$ ,  $s^*$  is different from  $s^n$  in the  $n^{\text{th}}$  digit. This contradicts the fact that  $f$  is a bijection. ■

**Theorem 1.36.** *The real interval  $[0, 1]$  (the set of real numbers between 0 and 1, inclusive) is uncountable.*

*Proof.* We will show that  $|[0, 1]| \geq |S|$ , where  $S$  is the same set as in the proof of Theorem 1.35. Treat each  $s = (d_1, d_2, \dots) \in S$  as the real number between 0 and 1 with the binary expansion  $0.d_1d_2\dots$ . Note that this does not establish a bijection; some real numbers have two binary expansions, e.g.,  $0.1 = 0.0111\dots$  (similarly, in decimal expansion, we have  $0.1 = 0.0999\dots$ <sup>2</sup>).

We may overcome this “annoyance” in two ways:

- Since each real number can have at most two decimal representations (most only have one), we can easily extend the above argument to show that  $|S| \leq |[0, 2]|$  (i.e., map  $[0, 1]$  to one representation, and  $[1, 2]$  to the other). It remains to show that  $|[0, 1]| = |[0, 2]|$  (can you think of a bijection here?).
- We may repeat Cantor’s diagonalization argument as in the proof of Theorem 1.35, in decimal expansion. When we construct  $s^*$ , avoid using the digits 9 and 0 (e.g., use only the digits 4 and 5). ■

A major open problem in mathematics (it was one of Hilbert’s 23 famous problems listed 1900) was whether there exists some set whose cardinality is between  $\mathbb{N}$  and  $\mathbb{R}$  (can you show that  $\mathbb{R}$  has the same cardinality as  $[0, 1]$ ?).

Here is a naive candidate:  $\mathcal{P}(\mathbb{N})$ . Unfortunately,  $\mathcal{P}(\mathbb{N})$  has the same cardinality as  $[0, 1]$ . Note that every element  $S \in \mathcal{P}(\mathbb{N})$  corresponds to an infinitely long sequence over digits  $\{0, 1\}$  (the  $n^{\text{th}}$  digit is 1 if and only if the number  $n \in S$ ). Again, we arrive at the set  $S$  in the proof of Theorem 1.35.

---

<sup>2</sup>For a proof, consider letting  $x = 0.0999\dots$ , and observe that  $10x - x = 0.999\dots - 0.0999\dots = 0.9$ , which solves to  $x = 0.1$ .

The Continuum Hypothesis states that no such set exists. Gödel and Cohen together showed (in 1940 and 1963) that this can neither be proved nor disproved using the standard axioms underlying mathematics (we will talk more about axioms when we get to logic).

# Proofs and Induction

*“Pics or it didn’t happen.”*  
– the internet

There are many forms of mathematical proofs. In this chapter we introduce several basic types of proofs, with special emphasis on a technique called *induction* that is invaluable to the study of discrete math.

## 2.1 Basic Proof Techniques

In this section we consider the following general task: given a premise  $X$ , how do we show that a conclusion  $Y$  holds? One way is to give a **direct proof**. Start with premise  $X$ , and directly deduce  $Y$  through a series of logical steps. See Claim 2.1 for an example.

**Claim 2.1.** *Let  $n$  be an integer. If  $n$  is even, then  $n^2$  is even. If  $n$  is odd, then  $n^2$  is odd.*

*Direct proof.* If  $n$  is even, then  $n = 2k$  for an integer  $k$ , and

$$n^2 = (2k)^2 = 4k^2 = 2 \cdot (2k^2), \text{ which is even.}$$

If  $n$  is odd, then  $n = 2k + 1$  for an integer  $k$ , and

$$n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2 \cdot (2k^2 + 2k) + 1, \text{ which is odd. } \blacksquare$$

There are also several forms of indirect proofs. A **proof by contrapositive** starts by assuming that the conclusion  $Y$  is false, and deduce that the premise  $X$  must also be false through a series of logical steps. See Claim 2.2 for an example.

**Claim 2.2.** *Let  $n$  be an integer. If  $n^2$  is even, then  $n$  is even.*

*Proof by contrapositive.* Suppose that  $n$  is not even. Then by Claim 2.1,  $n^2$  is not even as well. (Yes, the proof ends here.) ■

A **proof by contradiction**, on the other hand, assumes both that the premise  $X$  is true and the conclusion  $Y$  is false, and reach a logical fallacy. We give another proof of Claim 2.2 as example.

*Proof by contradiction.* Suppose that  $n^2$  is even, but  $n$  is odd. Applying Claim 2.1, we see that  $n^2$  must be odd. But  $n^2$  cannot be both odd and even! ■

In their simplest forms, it may seem that a direct proof, a proof by contrapositive, and a proof by contradiction may just be restatements of each other; indeed, one can always phrase a direct proof or a proof by contrapositive as a proof by contradiction (can you see how?). In more complicated proofs, however, choosing the “right” proof technique sometimes simplify or improve the aesthetics of a proof. Below is an interesting use of proof by contradiction.

**Theorem 2.3.**  $\sqrt{2}$  is irrational.

*Proof by contradiction.* Assume for contradiction that  $\sqrt{2}$  is rational. Then there exists integers  $p$  and  $q$ , with no common divisors, such that  $\sqrt{2} = p/q$  (i.e., the reduced fraction). Squaring both sides, we have:

$$2 = \frac{p^2}{q^2} \quad \Rightarrow \quad 2q^2 = p^2$$

This means  $p^2$  is even, and by Claim 2.2  $p$  is even as well. Let us replace  $p$  by  $2k$ . The expression becomes:

$$2q^2 = (2k)^2 = 4k^2 \quad \Rightarrow \quad q^2 = 2k^2$$

This time, we conclude that  $q^2$  is even, and so  $q$  is even as well. But this leads to a contradiction, since  $p$  and  $q$  now share a common factor of 2. ■

We end the section with the (simplest form of the) AM-GM inequality.

**Theorem 2.4** (Simple AM-GM inequality). *Let  $x$  and  $y$  be non-negative reals. Then,*

$$\frac{x+y}{2} \geq \sqrt{xy}$$

*Proof by contradiction.* Assume for contradiction that

$$\begin{aligned} & \frac{x+y}{2} < \sqrt{xy} \\ \Rightarrow & \frac{1}{4}(x+y)^2 < xy \quad \text{squaring non-negative values} \\ \Rightarrow & x^2 + 2xy + y^2 < 4xy \\ \Rightarrow & x^2 - 2xy + y^2 < 0 \\ \Rightarrow & (x-y)^2 < 0 \end{aligned}$$

But this is a contradiction since squares are always non-negative. ■

Note that the proof Theorem 2.4 can be easily turned into a direct proof; the proof of Theorem 2.3, on the other hand, cannot.

## 2.2 Proof by Cases and Examples

Sometimes the easiest way to prove a theorem is to split it into several cases.

**Claim 2.5.**  $(n+1)^2 \geq 2^n$  for all integers  $n$  satisfying  $0 \leq n \leq 5$ .

*Proof by cases.* There are only 6 different values of  $n$ . Let's try them all:

$n$	$(n+1)^2$	$2^n$
0	1	$\geq$ 1
1	4	$\geq$ 2
2	9	$\geq$ 4
3	16	$\geq$ 8
4	25	$\geq$ 16
5	36	$\geq$ 32

**Claim 2.6.** For all real  $x$ ,  $|x^2| = |x|^2$ .

*Proof by cases.* Split into two cases:  $x \geq 0$  and  $x < 0$ .

- If  $x \geq 0$ , then  $|x^2| = x^2 = |x|^2$ .
- If  $x < 0$ , then  $|x^2| = x^2 = (-x)^2 = |x|^2$ . ■

When presenting a proof by cases, make sure that *all* cases are covered! For some theorems, we only need to construct one case that satisfy the theorem statement.

**Claim 2.7.** Show that there exists some  $n$  such that  $(n+1)^2 \geq 2^n$ .

*Proof by example.*  $n = 6$ . ■

Sometimes we find a counterexample to disprove a theorem.

**Claim 2.8.** *Prove or disprove that  $(n+1)^2 \geq 2^n$  for all  $n \in \mathbb{N}$ .*

*Proof by (counter)example.* We choose to disprove the statement. Check out  $n = 6$ . Done. ■

The next proof does not explicitly construct the example asked by the theorem, but proves that such an example exists anyways. These type of proofs (among others) are *non-constructive*.

**Theorem 2.9.** *There exists irrational numbers  $x$  and  $y$  such that  $x^y$  is rational.*

*Non-constructive proof of existence.* We know  $\sqrt{2}$  is irrational from Theorem 2.3. Let  $z = \sqrt{2}^{\sqrt{2}}$ .

- If  $z$  is rational, then we are done ( $x = y = \sqrt{2}$ ).
- If  $z$  is irrational, then take  $x = z = \sqrt{2}^{\sqrt{2}}$ , and  $y = \sqrt{2}$ . Then:

$$x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$$

is indeed a rational number. ■

Here is another non-constructive existence proof. The game of *Chomp* is a 2-player game played on a “chocolate bar” made up of a rectangular grid. The players take turns to choose one block and “eat it” (remove from the board), together all other blocks that are below it or to its right (the whole lower right quadrant). The top left block is “poisoned” and the player who eats this loses.

**Theorem 2.10.** *Suppose the game of Chomp is played with rectangular grid strictly larger than  $1 \times 1$ . Player 1 (the first player) has a winning strategy.*

*Proof.* Consider following first move for player 1: eat the lower right most block. We have two cases<sup>1</sup>:

---

<sup>1</sup> Here we use the well-known fact of 2-player, deterministic, finite-move games without ties: any move is either a winning move (i.e., there is a strategy following this move that forces a win), or allows the opponent to follow up with a winning move. See Theorem 2.14 later for a proof of this fact.

- Case 1: There is a winning strategy for player 1 starting with this move. In this case we are done.
- Case 2: There is no winning strategy for player 1 starting with this move. In this case there is a winning strategy for player 2 following this move. But this winning strategy for player 2 is also a valid winning strategy for player 1, since the next move made by player 2 can be mimicked by player 1 (here we need the fact that the game is symmetric between the players). ■

While we have just shown that Player 1 can always win in a game of Chomp, no constructive strategy for Player 1 has been found for general rectangular grids (i.e., you cannot buy a strategy guide in store that tells you how to win Chomp). For a few specific cases though, we do know good strategies for Player 1. E.g., given a  $n \times n$  square grid, Player 1 starts by removing a  $n - 1 \times n - 1$  (unique) block, leaving an L-shaped piece of chocolate with two “arms”; thereafter, Player 1 simply mirrors Player 2’s move, i.e., whenever Player 2 takes a bite from one of the arms, Player 1 takes the same bite on the other arm.

As our last example, consider tiling a  $8 \times 8$  chess board with dominoes ( $2 \times 1$  pieces), i.e., the whole board should be covered by dominoes without any dominoes overlapping each other or sticking out.

**Q:** Can we tile it?

**A:** Yes. Easy to give a proof by example (constructive existence proof).

**Q:** What if I remove one grid of the check board?

**A:** No. Each domino covers 2 grids, so the number of covered grids is always even, but the board has 63 pieces (direct proof / proof by contradiction).

**Q:** What if I remove the top left and bottom right grids?

**A:** No. Each domino covers 1 grid of each colors. The top left and bottom right grids have the same color, however, so the remaining board has more white grids than black (or more black grids than white) (direct proof / proof by contradiction).

## 2.3 Induction

We start with the most basic form of induction: induction over the natural numbers. Suppose we want to show that a statement is true for all natural

numbers, e.g., for all  $n$ ,  $1 + 2 + \cdots + n = n(n + 1)/2$ . The basic idea is to approach the proof in two steps:

1. First prove that the statement is true for  $n = 1$ . This is called the **base case**.
2. Next prove that whenever the statement is true for case  $n$ , then it is also true for case  $n + 1$ . This is called the **inductive step**.

The base case shows that the statement is true for  $n = 1$ . Then, by repeatedly applying the inductive step, we see that the statement is true for  $n = 2$ , and then  $n = 3$ , and then  $n = 4, 5, \dots$ ; we just covered all the natural numbers! Think of pushing over a long line of dominoes. The induction step is just like setting up the dominoes; we make sure that if a domino falls, so will the next one. The base case is then analogous to pushing down the first domino. The result? All the dominoes fall.

Follow these steps to write an inductive proof:

1. Start by formulating the **inductive hypothesis** (i.e., what you want to prove). It should be parametrized by a natural number. E.g.,  $P(n) : 1 + 2 + \cdots + n = n(n + 1)/2$ .
2. Show that  $P(\text{base})$  is true for some appropriate base case. Usually  $\text{base}$  is 0 or 1.
3. Show that the inductive step is true, i.e., assume  $P(n)$  holds and prove that  $P(n + 1)$  holds as well.

Violà, we have just shown that  $P(n)$  holds for all  $n \geq \text{base}$ . Note that the base case does not always have to be 0 or 1; we can start by showing that something is  $P(n)$  is true for  $n = 5$ ; this combined with the inductive step shows that  $P(n)$  is true for all  $n \geq 5$ . Let's put our new found power of inductive proofs to the test!

**Claim 2.11.** *For all positive integers  $n$ ,  $1 + 2 + \cdots + n = n(n + 1)/2$ .*

*Proof.* Define out induction hypothesis  $P(n)$  to be true if

$$\sum_{i=1}^n i = \frac{1}{2}n(n + 1)$$

**Base case:**  $P(1)$  is clearly true by inspection.

**Inductive Step:** Assume  $P(n)$  is true; we wish to show that  $P(n + 1)$  is true as well:

$$\begin{aligned} \sum_{i=1}^{n+1} i &= \left( \sum_{i=1}^n i \right) + (n + 1) \\ &= \frac{1}{2} n(n + 1) + n + 1 && \text{using } P(n) \\ &= \frac{1}{2} (n(n + 1) + 2(n + 1)) = \frac{1}{2} ((n + 1)(n + 2)) \end{aligned}$$

This is exactly  $P(n + 1)$ . ■

**Claim 2.12.** *For any finite set  $S$ ,  $|\mathcal{P}(S)| = 2^{|S|}$ .*

*Proof.* Define our induction hypothesis  $P(n)$  to be true if for every finite set  $S$  of cardinality  $|S| = n$ ,  $|\mathcal{P}(S)| = 2^n$ .

**Base case:**  $P(0)$  is true since the only finite set of size 0 is the empty set  $\emptyset$ , and the power set of the empty set,  $\mathcal{P}(\emptyset) = \{\emptyset\}$ , has cardinality 1.

**Inductive Step:** Assume  $P(n)$  is true; we wish to show that  $P(n + 1)$  is true as well. Consider a finite set  $S$  of cardinality  $n + 1$ . Pick an element  $e \in S$ , and consider  $S' = S - \{e\}$ . By the induction hypothesis,  $|\mathcal{P}(S')| = 2^n$ .

Now consider  $\mathcal{P}(S)$ . Observe that a set in  $\mathcal{P}(S)$  either contains  $e$  or not; furthermore, there is a one-to-one correspondence between the sets containing  $e$  and the sets not containing  $e$  (can you think of the bijection?). We have just partitioned  $\mathcal{P}(S)$  into two equal cardinality subsets, one of which is  $\mathcal{P}(S')$ . Therefore  $|\mathcal{P}(S)| = 2|\mathcal{P}(S')| = 2^{n+1}$ . ■

**Claim 2.13.** *The following two properties of graphs are equivalent (recall that these are the definitions of transitivity on the graph of a relation):*

1. *For any three nodes  $x$ ,  $y$  and  $z$  such that there is an edge from  $x$  to  $y$  and from  $y$  to  $z$ , there exist an edge from  $x$  to  $z$ .*
2. *Whenever there is a path from node  $x$  to node  $y$ , there is also a direct edge from  $x$  to  $y$ .*

*Proof.* Clearly property 2 implies property 1. We use induction to show that property 1 implies property 2 as well. Let  $G$  be a graph on which property 1 holds. Define our induction hypothesis  $P(n)$  to be true if for every path of length  $n$  in  $G$  from node  $x$  to node  $y$ , there exists a direct edge from  $x$  to  $y$ .

**Base case:**  $P(1)$  is simply true (path of length 1 is already a direct edge).

**Inductive Step:** Assume  $P(n)$  is true; we wish to show that  $P(n + 1)$  is true as well. Consider a path of length  $n + 1$  from node  $x$  to node  $y$ , and let

$z$  be the first node after  $x$  on the path. We now have a path of length  $n$  from node  $z$  to  $y$ , and by the induction hypothesis, a direct edge from  $z$  to  $y$ . Now that we have a directly edge from  $x$  to  $z$  and from  $z$  to  $y$ , property 1 implies that there is a direct edge from  $x$  to  $y$ . ■

**Theorem 2.14.** *In a deterministic, finite 2-player game of perfect information without ties, either player 1 or player 2 has a winning strategy, i.e., a strategy that guarantees a win.<sup>2,3</sup>*

*Proof.* Let  $P(n)$  be the theorem statement for  $n$ -move games.

**Base case:**  $P(1)$  is trivially true. Since only player 1 gets to move, if there exists some move that makes player 1 win, then player 1 has a winning strategy; otherwise player 2 always wins and has a winning strategy (the strategy of doing nothing).

**Inductive Step:** Assume  $P(n)$  is true; we wish to show that  $P(n + 1)$  is true as well. Consider some  $n + 1$ -move game. After player 1 makes the first move, we end up in a  $n$ -move game. Each such game has a winning strategy for either player 1 or player 2 by  $P(n)$ .

- If all these games have a winning strategy for player 2<sup>4</sup>, then no matter what move player 1 plays, player 2 has a winning strategy
- If one these games have a winning strategy for player 1, then player 1 has a winning strategy (by making the corresponding first move). ■

In the next example, induction is used to prove only a subset of the theorem to give us a jump start; the theorem can then be completed using other techniques.

**Theorem 2.15** (AM-GM Inequality). *Let  $x_1, x_2, \dots, x_n$  be a sequence of non-negative reals. Then*

$$\frac{1}{n} \sum_i x_i \geq \left( \prod_i x_i \right)^{1/n}$$

---

<sup>3</sup>By deterministic, we mean the game has no randomness and depends on only on player moves (e.g., not backgammon). By finite, we mean the game is always ends in some predetermined fix number of moves; in chess, even though there are infinite sequences of moves that avoid both checkmates and stalemates, many draw rules (e.g., cannot have more than 100 consecutive moves without captures or pawn moves) ensures that chess is a finite game. By perfect information, we mean that both players knows each other's past moves (e.g., no fog of war).

<sup>4</sup> By this we mean the player 1 of the  $n$ -move game (the next player to move) has a winning strategy

*Proof.* In this proof we use the notation

$$\text{AM}(x_1, \dots, x_n) = \frac{1}{n} \sum_{i=1}^n x_i \quad \text{GM}(x_1, \dots, x_n) = \left( \prod_{i=1}^n x_i \right)^{1/n}$$

Let us first prove the AM-GM inequality for values of  $n = 2^k$ . Define our induction hypothesis  $P(k)$  to be true if AM-GM holds for  $n = 2^k$ .

**Base case:**  $P(0)$  (i.e.,  $n = 1$ ) trivially holds, and  $P(1)$  (i.e.,  $n = 2$ ) was shown in Theorem 2.4.

**Inductive Step:** Assume  $P(k)$  is true; we wish to show that  $P(k+1)$  is true as well. Given a sequence of length  $2^{k+1}$ ,  $\vec{X} = (x_1, \dots, x_{2^{k+1}})$ , we split it into two sequences  $\vec{X}_1 = (x_1, \dots, x_{2^k})$ ,  $\vec{X}_2 = (x_{2^k+1}, x_{2^k+2}, \dots, x_{2^{k+1}})$ . Then:

$$\begin{aligned} \text{AM}(\vec{X}) &= \frac{1}{2}(\text{AM}(\vec{X}_1) + \text{AM}(\vec{X}_2)) \\ &\geq \frac{1}{2}(\text{GM}(\vec{X}_1) + \text{GM}(\vec{X}_2)) \quad \text{by the induction hypothesis } P(k) \\ &= \text{AM}(\text{GM}(\vec{X}_1), \text{GM}(\vec{X}_2)) \\ &\geq \text{GM}(\text{GM}(\vec{X}_1), \text{GM}(\vec{X}_2)) \quad \text{by Theorem 2.4, i.e., } P(1) \\ &= \left( \left( \prod_{i=1}^{2^k} x_i \right)^{\frac{1}{2^k}} \left( \prod_{i=2^k+1}^{2^{k+1}} x_i \right)^{\frac{1}{2^k}} \right)^{1/2} \\ &= \left( \prod_{i=1}^{2^{k+1}} x_i \right)^{\frac{1}{2^{k+1}}} = \text{GM}(\vec{X}) \end{aligned}$$

We are now ready to show the AM-GM inequality for sequences of all lengths. Given a sequence  $\vec{X} = (x_1, \dots, x_n)$  where  $n$  is not a power of 2, find the smallest  $k$  such that  $2^k > n$ . Let  $\alpha = \text{AM}(\vec{X})$ , and consider a new sequence

$$\vec{X}' = (x_1, \dots, x_n, x_{n+1} = \alpha, x_{n+2} = \alpha, \dots, x_{2^k} = \alpha)$$

and verify that  $\text{AM}(\vec{X}') = \text{AM}(\vec{X}) = \alpha$ . Apply  $P(k)$  (the AM-GM inequality for sequences of length  $2^k$ ), we have:

$$\begin{aligned}
 \text{AM}(\vec{X}') &= \alpha \geq \text{GM}(\vec{X}') = \left( \prod_{i=1}^{2^k} x_i \right)^{1/2^k} \\
 \Rightarrow \quad \alpha^{2^k} &\geq \prod_{i=1}^{2^k} x_i = \prod_{i=1}^n x_i \cdot \alpha^{2^k-n} \\
 \Rightarrow \quad \alpha^n &\geq \prod_{i=1}^n x_i \\
 \Rightarrow \quad \alpha &\geq \left( \prod_{i=1}^n x_i \right)^{1/n} = \text{GM}(\vec{X})
 \end{aligned}$$

This finishes our proof (recalling that  $\alpha = \text{AM}(\vec{X})$ ). ■

Note that for the inductive proof in Theorem 2.15, we needed to show both base cases  $P(0)$  and  $P(1)$  to avoid circular arguments, since the inductive step relies on  $P(1)$  to be true.

A common technique in inductive proofs is to define a *stronger* induction hypothesis than is needed by the theorem. A stronger induction hypothesis  $P(n)$  sometimes make the induction step simpler, since we would start each induction step with a stronger premise. As an example, consider the game of “coins on the table”. The game is played on a round table between two players. The players take turns putting on one penny at a time onto the table, without overlapping with previous pennies; the first player who cannot add another coin losses.

**Theorem 2.16.** *The first player has a winning strategy in the game of “coins on the table”.*

*Proof.* Consider the following strategy for player 1 (the first player). Start first by putting a penny centered on the table, and in all subsequent moves, simply mirror player 2’s last move (i.e., place a penny diagonally opposite of player 2’s last penny). We prove by induction that player 1 can always put down a coin, and therefore will win eventually (when the table runs out of space).

Define the induction hypothesis  $P(n)$  to be true if on the  $n^{\text{th}}$  move of player 1, player 1 can put down a penny according to its strategy, and leave the table symmetric about the centre (i.e., looks the same if rotated 180 degrees).

**Base case:**  $P(1)$  holds since player 1 can always start by putting one penny at the centre of the table, leaving the table symmetric.

**Inductive Step:** Assume  $P(n)$  is true; we wish to show that  $P(n + 1)$  is true as well. By the induction hypothesis, after player 1's  $n^{\text{th}}$  move, the table is symmetric. Therefore, if player 2 now puts down a penny, the diagonally opposite spot must be free of pennies, allowing player 1 to set down a penny as well. Moreover, after player 1's move, the table is back to being symmetric. ■

The Towers of Hanoi is a puzzle game where there are three poles, and a number of increasingly larger rings that are originally all stacked in order of size on the first pole, largest at the bottom. The goal of the puzzle is to move all the rings to another pole (pole 2 or pole 3), with the rule that:

- You may only move one ring a time, and it must be the top most ring in one of the three potential stacks.
- At any point, no ring may be placed on top of a smaller ring.<sup>5</sup>

**Theorem 2.17.** *The Towers of Hanoi with  $n$  rings can be solved in  $2^n - 1$  moves.*

*Proof.* Define the induction hypothesis  $P(n)$  to be true if the theorem statement is true for  $n$  rings.

**Base case:**  $P(1)$  is clearly true. Just move the ring.

**Inductive Step:** Assume  $P(n)$  is true; we wish to show that  $P(n + 1)$  is true as well. Number the rings 1 to  $n + 1$ , from smallest to largest (top to bottom on the original stack). First move rings 1 to  $n$  from pole 1 to pole 2; this takes  $2^n - 1$  steps by the induction hypothesis  $P(n)$ . Now move ring  $n + 1$  from pole 1 to pole 3. Finally, move rings 1 to  $n$  from pole 2 to pole 3; again, this takes  $2^n - 1$  steps by the induction hypothesis  $P(n)$ . In total we have used  $(2^n - 1) + 1 + (2^n - 1) = 2^{n+1} - 1$  moves. (Convince yourself that this recursive definition of moves will never violate the rule that no ring may be placed on top of a smaller ring.) ■

Legends say that such a puzzle was found in a temple with  $n = 64$  rings, left for the priests to solve. With our solution, that would require  $2^{64} - 1 \approx 1.8 \times 10^{19}$  moves. Is our solution just silly and takes too many moves?

**Theorem 2.18.** *The Towers of Hanoi with  $n$  rings requires at least  $2^n - 1$  moves to solve. Good luck priests!*

---

<sup>5</sup>Squashing small rings with large rings is bad, m'kay?

*Proof.* Define the induction hypothesis  $P(n)$  to be true if the theorem statement is true for  $n$  rings.

**Base case:**  $P(1)$  is clearly true. You need to move the ring.

**Inductive Step:** Assume  $P(n)$  is true; we wish to show that  $P(n + 1)$  is true as well. Again we number the rings 1 to  $n + 1$ , from smallest to largest (top to bottom on the original stack). Consider ring  $n + 1$ . It needs to be moved at some point. Without loss of generality, assume its final destination is pole 3. Let the  $k^{\text{th}}$  move be the first move where ring  $n + 1$  is moved away from pole 1 (to pole 2 or 3), and let the  $k'^{\text{th}}$  move be the last move where ring  $n + 1$  is moved to pole 3 (away from pole 1 to pole 2),

Before performing move  $k$ , all  $n$  other rings must first be moved to the remaining free pole (pole 3 or 2); by the induction hypothesis  $P(n)$ ,  $2^n - 1$  steps are required before move  $k$ . Similarly, after performing move  $k$ , all  $n$  other rings must be on the remaining free pole (pole 2 or 1); by the induction hypothesis  $P(n)$ ,  $2^n - 1$  steps are required after move  $k'$  to complete the puzzle. In the best case where  $k = k'$  (i.e., they are the same move), we still need at least  $(2^n - 1) + 1 + (2^n - 1) = 2^{n+1} - 1$  moves. ■

### Strong Induction

Taking the dominoes analogy one step further, a large domino may require the combined weight of all the previous toppling over before it topples over as well. The mathematical equivalent of this idea is *strong induction*. To prove that a statement  $P(n)$  is true for (a subset of) positive integers, the basic idea is:

1. First prove that  $P(n)$  is true for some base values of  $n$  (e.g.,  $n = 1$ ). These are the **base cases**.
2. Next prove that if  $P(k)$  is true for  $1 \leq k \leq n$ , then  $P(n + 1)$  is true. This is called the **inductive step**.

How many base cases do we need? It roughly depends on the following factors:

- What is the theorem? Just like basic induction, if we only need  $P(n)$  to be true for  $n \geq 5$ , then we don't need base cases  $n < 5$ .
- What does the induction hypothesis need? Often to show  $P(n + 1)$ , instead of requiring that  $P(k)$  be true for  $1 \leq k \leq n$ , we actually need, say  $P(n)$  and  $P(n - 1)$  to be true. Then having the base case of  $P(1)$  isn't enough for the induction hypothesis to prove  $P(3)$ ;  $P(2)$  is another required base case.

Let us illustrate both factors with an example.

**Claim 2.19.** Suppose we have an unlimited supply of 3 cent and 5 cent coins. Then we can pay any amount  $\geq 8$  cents.

*Proof.* Let  $P(n)$  be the true if we can indeed form  $n$  cents with 3 cent and 5 cent coins.

**Base case:**  $P(8)$  is true since  $3 + 5 = 8$ .

**Inductive Step:** Assume  $P(k)$  is true for  $8 \leq k \leq n$ ; we wish to show that  $P(n+1)$  is true as well. This seems easy; if  $P(n-2)$  is true, then adding another 3 cent coin gives us  $P(n+1)$ . But the induction hypothesis doesn't necessarily say  $P(n-2)$  is true! For  $(n+1) \geq 11$ , the induction hypothesis does apply (since  $n-2 \geq 8$ ). For  $n+1 = 9$  or 10, we have to do more work.

**Additional base cases:**  $P(9)$  is true since  $3 + 3 + 3 = 9$ , and  $P(10)$  is true since  $5 + 5 = 10$ . ■

With any induction, especially strong induction, it is *very important* to check for sufficient base cases! Here is what might happen in a faulty strong inductive proof.<sup>6</sup> Let  $P(n)$  be true if for all groups of  $n$  women, whenever one woman is blonde, then all of the women are blonde; since there is at least one blonde in the world, once I am done with the proof, every women in the world will be blonde!

**Base case:**  $P(1)$  is clearly true.

**Induction step:** Suppose  $P(k)$  is true for all  $1 \leq k \leq n$ ; we wish to show  $P(n+1)$  is true as well. Given a set  $W$  of  $n+1$  women in which  $x \in W$  is blonde, take any two strict subsets  $A, B \subsetneq W$  (in particular  $|A|, |B| < n+1$ ) such that they both contain the blonde ( $x \in A, x \in B$ ), and  $A \cup B = W$  (no one is left out). Applying the induction hypothesis to  $A$  and  $B$ , we conclude that all the women in  $A$  and  $B$  are blonde, and so everyone in  $W$  is blonde.

What went wrong?<sup>7</sup>

---

<sup>6</sup>Another example is to revisit Claim 2.19. If we use the same proof to show that  $P(n)$  is true for all  $n \geq 3$ , without the additional base cases, the proof will be "seemingly correct". What is the obvious contradiction?

<sup>7</sup>Hint: Can you trace the argument when  $n = 2$ ?

## 2.4 Inductive Definitions

In addition to being a proof technique, induction can be used to *define* mathematical objects. Some basic examples include products or sums of sequences:

- The factorial function  $n!$  over non-negative integers can be formally defined by

$$0! = 1; \quad (n+1)! = n! \cdot (n+1)$$

- The cumulative sum of a sequence  $x_1, \dots, x_k$ , often written as  $S(n) = \sum_{i=1}^n x_i$ , can be formally defined by

$$S(0) = 0; \quad S(n+1) = S(n) + x_{n+1}$$

Just like inductive proofs, inductive definitions start with a “base case” (e.g., defining  $0! = 1$ ), and has an “inductive step” to define the rest of the values (e.g., knowing  $0! = 1$ , we can compute  $1! = 1 \cdot 1 = 1$ ,  $2! = 1 \cdot 2 = 2$ , and so on).

### Recurrence Relations

When an inductive definition generates a sequence (e.g., the factorial sequence is  $1, 1, 2, 6, 24, \dots$ ), we call the definition a *recurrence relation*. We can generalize inductive definitions and recurrence relations in a way much like we generalize inductive proofs with strong induction. For example, consider a sequence defined by:

$$a_0 = 1; \quad a_1 = 2; \quad a_n = 4a_{n-1} - 4a_{n-2}$$

According to the definition, the next few terms in the sequence will be

$$a_2 = 4; \quad a_3 = 8$$

At this point, the sequence looks suspiciously as if  $a_n = 2^n$ . Let’s prove this by induction!

*Proof.* Define  $P(n)$  to be true if  $a_n = 2^n$ .

**Base case:**  $P(0)$  and  $P(1)$  are true since  $a_0 = 1 = 2^0$ ,  $a_1 = 2 = 2^1$ .

**Inductive Step:** Assume  $P(k)$  is true for  $0 \leq k \leq n$ ; we wish to show that  $P(n+1)$  is true as well for  $n+1 \geq 2$ . We have

$$\begin{aligned} a_{n+1} &= 4a_n - 4a_{n-1} \\ &= 4 \cdot 2^n - 4 \cdot 2^{n-1} && \text{by } P(n) \text{ and } P(n-1) \\ &= 2^{n+2} - 2^{n+1} = 2^{n+1} \end{aligned}$$

This is exactly  $P(n+1)$ . ■

Remember that it is very important to check the *all* the base cases (especially since this proof uses strong induction). Let us consider another example:

$$b_0 = 1; \quad b_1 = 1; \quad b_n = 4b_{n-1} - 3b_{n-2}$$

From the recurrence part of the definition, it looks like the sequence  $(b_n)_n$  will eventually out grow the sequence  $(a_n)_n$ . Based only on this intuition, let us conjecture that  $b_n = 3^n$ .

*Possibly correct proof.* Define  $P(n)$  to be true if  $b_n = 3^n$ .

**Base case:**  $P(0)$  is true since  $b_0 = 1 = 3^0$ .

**Inductive Step:** Assume  $P(k)$  is true for  $0 \leq k \leq n$ ; we wish to show that  $P(n+1)$  is true as well for  $n+1 \geq 3$ . We have

$$\begin{aligned} b_{n+1} &= 4b_n - 3b_{n-1} \\ &= 4 \cdot 3^n - 3 \cdot 3^{n-1} && \text{by } P(n) \text{ and } P(n-1) \\ &= (3^{n+1} + 3^n) - 3^n = 3^{n+1} \end{aligned} \quad \blacksquare$$

Wow! Was that a lucky guess or what. Let us actually compute a few terms of  $(b_n)_n$  to make sure...

$$\begin{aligned} b_2 &= 4b_1 - 3b_0 = 4 - 3 = 1, \\ b_3 &= 4b_2 - 3b_1 = 4 - 3 = 1, \\ &\vdots \quad \text{:(} \end{aligned}$$

Looks like in fact,  $b_n = 1$  for all  $n$  (as an exercise, prove this by induction). What went wrong with our earlier “proof”? Note that  $P(n-1)$  is only well defined if  $n \geq 1$ , so the inductive step does not work when we try to show  $P(1)$  (when  $n = 0$ ). As a result we need an extra base case to handle  $P(1)$ ; a simple check shows that it is just not true:  $b_1 = 1 \neq 3^1 = 3$ . (On the other hand, if we define  $b'_0 = 1$ ,  $b'_1 = 3$ , and  $b'_n = 4b'_{n-1} - 3b'_{n-2}$ , then we can recycle our “faulty proof” and show that  $b'_n = 3^n$ ).

In the examples so far, we guessed at a closed form formula for the sequences  $(a_n)_n$  and  $(b_n)_n$ , and then proved that our guesses were correct using induction. For certain recurrence relations, there are direct methods for computing a closed form formula of the sequence.

**Theorem 2.20.** Consider the recurrence relation  $a_n = c_1 a_{n-1} + c_2 a_{n-2}$  with  $c_2 \neq 0$ , and arbitrary base cases for  $a_0$  and  $a_1$ . Suppose that the polynomial  $x^2 - (c_1x + c_2)$  has two distinct roots  $r_1$  and  $r_2$  (these roots are non-zero since  $c_2 \neq 0$ ). Then there exists constants  $\alpha$  and  $\beta$  such that  $a_n = \alpha x_1^n + \beta x_2^n$ .

*Proof.* The polynomial  $f(x) = x^2 - (c_1x + c_2)$  is called the *characteristic polynomial* for the recurrence relation  $a_n = c_1a_{n-1} + c_2a_{n-2}$ . Its significance can be explained by the sequence  $(r^0, r^1, \dots)$  where  $r$  is a root of  $f(x)$ ; we claim that this sequence satisfies the recurrence relation (with base cases set as  $r^0$  and  $r^1$ ). Let  $P(n)$  be true if  $a_n = r^n$ .

**Inductive Step:** Assume  $P(k)$  is true for  $0 \leq k \leq n$ ; we wish to show that  $P(n+1)$  is true as well. Observe that:

$$\begin{aligned} a_{n+1} &= c_1a_n + c_2a_{n-1} \\ &= c_1r^n + c_2r^{n-1} && \text{by } P(n-1) \text{ and } P(n) \\ &= r^{n-1}(c_1r + c_2) \\ &= r^{n-1} \cdot r^2 && \text{since } r \text{ is a root of } f(x) \\ &= r^{n+1} \end{aligned}$$

Recall that there are two distinct roots,  $r_1$  and  $r_2$ , so we actually have two sequences that satisfy the recurrence relation (under proper base cases). In fact, because the recurrence relation is *linear* ( $a_n$  depends linearly on  $a_{n-1}$  and  $a_{n-2}$ ), and *homogeneous* (there is no constant term in the recurrence relation), any sequence of the form  $a_n = \alpha r_1^n + \beta r_2^n$  will satisfy the recurrence relation; (this can be shown using a similar inductive step as above).

Finally, does sequences of the form  $a_n = \alpha r_1^n + \beta r_2^n$  cover all possible base cases? The answer is yes. Given any base case  $a_0 = a_0^*$ ,  $a_1 = a_1^*$ , we can solve for the unique value of  $\alpha$  and  $\beta$  using the linear system:

$$\begin{aligned} a_0^* &= \alpha r_1^0 + \beta r_2^0 = \alpha + \beta \\ a_1^* &= \alpha r_1^1 + \beta r_2^1 = \alpha r_1 + \beta r_2 \end{aligned}$$

The studious reader should check that this linear system always has a unique solution (say, by checking that the determinant of the system is non-zero). ■

The technique outlined in Theorem 2.20 can be extended to any recurrence relation of the form

$$a_n = c_1a_{n-1} + c_2a_{n-2} + \cdots + c_ka_{n-k}$$

for some constant  $k$ ; the solution is always a linear combination of  $k$  sequences of the form  $(r^0, r^1, r^2, \dots)$ , one for each distinct root  $r$  of the characteristic polynomial

$$f(x) = x^k - (c_1x^{k-1} + c_2x^{k-2} + \cdots + c_k)$$

In the case that  $f(x)$  has duplicate roots, say when a root  $r$  has multiplicity  $m$ , in order to still have a total of  $k$  distinct sequences, we associate the following  $m$  sequences with  $r$ :

$$\begin{aligned} & (r^0, r^1, r^2, \dots, r^n, \dots) \\ & (0 \cdot r^0, 1 \cdot r^1, 2 \cdot r^2, \dots, n r^n, \dots) \\ & (0^2 \cdot r^0, 1^2 \cdot r^1, 2^2 \cdot r^2, \dots, n^2 r^n, \dots) \\ & \quad \vdots \\ & (0^{m-1} \cdot r^0, 1^{m-1} \cdot r^1, 2^{m-1} \cdot r^2, \dots, n^{m-1} r^n, \dots) \end{aligned}$$

For example, if  $f(x)$  has degree 2 and has a unique root  $r$  with multiplicity 2, then the general form solution to the recurrence is

$$a_n = \alpha r^n + \beta n r^n$$

We omit the proof of this general construction. Interestingly, the same technique is used in many other branches of mathematics (for example, to solve linear ordinary differential equations).

As an example, let us derive a closed form expression to the famous Fibonacci numbers.

**Theorem 2.21.** Define the Fibonacci sequence inductively as

$$f_0 = 0; \quad f_1 = 1; \quad f_n = f_{n-1} + f_{n-2}$$

Then

$$f_n = \frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1-\sqrt{5}}{2} \right)^n \quad (2.1)$$

*Proof.* It is probably hard to guess (2.1); we will derive it from scratch. The characteristic polynomial here is  $f(x) = x^2 - (x + 1)$ , which has roots

$$\frac{1+\sqrt{5}}{2}, \quad \frac{1-\sqrt{5}}{2}$$

This means the Fibonacci sequence can be expressed as

$$f_n = \alpha \left( \frac{1+\sqrt{5}}{2} \right)^n + \beta \left( \frac{1-\sqrt{5}}{2} \right)^n$$

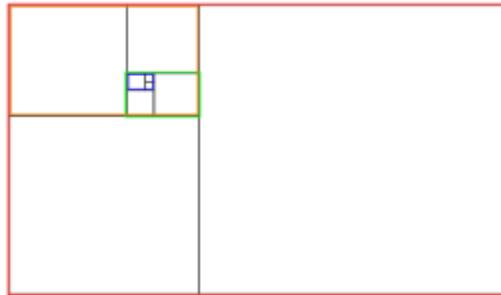


Figure 2.1: Approximating the golden ratio with rectangles whose side lengths are consecutive elements of the Fibonacci sequence. Do the larger rectangles look more pleasing than the smaller rectangles to you?

for constants  $\alpha$  and  $\beta$ . Substituting  $f_0 = 0$  and  $f_1 = 1$  gives us

$$\begin{aligned} 0 &= \alpha + \beta \\ 1 &= \alpha \left( \frac{1 + \sqrt{5}}{2} \right) + \beta \left( \frac{1 - \sqrt{5}}{2} \right) \end{aligned}$$

which solves to  $\alpha = 1/\sqrt{5}$ ,  $\beta = -1/\sqrt{5}$ . ■

As a consequence of (2.1), we know that for large  $n$ ,

$$f_n \approx \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^n$$

because the other term approaches zero. This in turn implies that

$$\lim_{n \rightarrow \infty} \frac{f_{n+1}}{f_n} = \frac{1 + \sqrt{5}}{2}$$

which is the **golden ratio**. It is widely believed that a rectangle whose ratio (length divided by width) is golden is pleasing to the eye; as a result, the golden ratio can be found in many artworks and architectures throughout history.

## Exercises

1. Use a direct proof to show that the sum of two odd integers is even.
2. Use a direct proof to show that the sum of two even integers is even.
3. Show that the square of an even number is an even number using a direct proof.
4. Show that the additive inverse, or negative, of an even number is an even number using a direct proof.
5. Prove that if  $m + n$  and  $n + p$  are even integers, where  $m$ ,  $n$ , and  $p$  are integers, then  $m + p$  is even. What kind of proof did you use?
6. Use a direct proof to show that the product of two odd numbers is odd.
7. Use a direct proof to show that every odd integer is the difference of two squares.
8. Prove that if  $n$  is a perfect square, then  $n + 2$  is not a perfect square.
9. Use a proof by contradiction to prove that the sum of an irrational number and a rational number is irrational.
10. Use a direct proof to show that the product of two rational numbers is rational.
11. Prove or disprove that the product of two irrational numbers is irrational.
12. Prove or disprove that the product of a nonzero rational number and an irrational number is irrational.
13. Prove that if  $x$  is irrational, then  $1/x$  is irrational.
14. Prove that if  $x$  is rational and  $x \neq 0$ , then  $1/x$  is rational.
15. Use a proof by contraposition to show that if  $x + y \geq 2$ , where  $x$  and  $y$  are real numbers, then  $x \geq 1$  or  $y \geq 1$ .
16. Prove that if  $m$  and  $n$  are integers and  $mn$  is even, then  $m$  is even or  $n$  is even.
17. Show that if  $n$  is an integer and  $n^3 + 5$  is odd, then  $n$  is even using
  - a proof by contraposition.
  - a proof by contradiction.
18. Prove that if  $n$  is an integer and  $3n + 2$  is even, then  $n$  is even using
  - a proof by contraposition.
  - a proof by contradiction.
19. Prove the proposition  $P(0)$ , where  $P(n)$  is the proposition "If  $n$  is a positive integer greater than 1, then  $n^2 > n$ ." What kind of proof did you use?
20. Prove the proposition  $P(1)$ , where  $P(n)$  is the proposition "If  $n$  is a positive integer, then  $n^2 \geq n$ ." What kind of proof did you use?
21. Let  $P(n)$  be the proposition "If  $a$  and  $b$  are positive real numbers, then  $(a + b)^n \geq a^n + b^n$ ." Prove that  $P(1)$  is true. What kind of proof did you use?
22. Show that if you pick three socks from a drawer containing just blue socks and black socks, you must get either a pair of blue socks or a pair of black socks.
23. Show that at least ten of any 64 days chosen must fall on the same day of the week.
24. Show that at least three of any 25 days chosen must fall in the same month of the year.
25. Use a proof by contradiction to show that there is no rational number  $r$  for which  $r^3 + r + 1 = 0$ . [Hint: Assume that  $r = a/b$  is a root, where  $a$  and  $b$  are integers and  $a/b$  is in lowest terms. Obtain an equation involving integers by multiplying by  $b^3$ . Then look at whether  $a$  and  $b$  are each odd or even.]
26. Prove that if  $n$  is a positive integer, then  $n$  is even if and only if  $7n + 4$  is even.
27. Prove that if  $n$  is a positive integer, then  $n$  is odd if and only if  $5n + 6$  is odd.
28. Prove that  $m^2 = n^2$  if and only if  $m = n$  or  $m = -n$ .
29. Prove or disprove that if  $m$  and  $n$  are integers such that  $mn = 1$ , then either  $m = 1$  and  $n = 1$ , or else  $m = -1$  and  $n = -1$ .
30. Show that these three statements are equivalent, where  $a$  and  $b$  are real numbers: (i)  $a$  is less than  $b$ , (ii) the average of  $a$  and  $b$  is greater than  $a$ , and (iii) the average of  $a$  and  $b$  is less than  $b$ .
31. Show that these statements about the integer  $x$  are equivalent: (i)  $3x + 2$  is even, (ii)  $x + 5$  is odd, (iii)  $x^2$  is even.
32. Show that these statements about the real number  $x$  are equivalent: (i)  $x$  is rational, (ii)  $x/2$  is rational, (iii)  $3x - 1$  is rational.
33. Show that these statements about the real number  $x$  are equivalent: (i)  $x$  is irrational, (ii)  $3x + 2$  is irrational, (iii)  $x/2$  is irrational.
34. Is this reasoning for finding the solutions of the equation  $\sqrt{2x^2 - 1} = x$  correct? (1)  $\sqrt{2x^2 - 1} = x$  is given; (2)  $2x^2 - 1 = x^2$ , obtained by squaring both sides of (1); (3)  $x^2 - 1 = 0$ , obtained by subtracting  $x^2$  from both sides of (2); (4)  $(x - 1)(x + 1) = 0$ , obtained by factoring the left-hand side of  $x^2 - 1$ ; (5)  $x = 1$  or  $x = -1$ , which follows because  $ab = 0$  implies that  $a = 0$  or  $b = 0$ .
35. Are these steps for finding the solutions of  $\sqrt{x+3} = 3 - x$  correct? (1)  $\sqrt{x+3} = 3 - x$  is given; (2)  $x + 3 = x^2 - 6x + 9$ , obtained by squaring both sides of (1); (3)  $0 = x^2 - 7x + 6$ , obtained by subtracting  $x + 3$  from both sides of (2); (4)  $0 = (x - 1)(x - 6)$ , obtained by factoring the right-hand side of (3); (5)  $x = 1$  or  $x = 6$ , which follows from (4) because  $ab = 0$  implies that  $a = 0$  or  $b = 0$ .
36. Show that the propositions  $p_1$ ,  $p_2$ ,  $p_3$ , and  $p_4$  can be shown to be equivalent by showing that  $p_1 \leftrightarrow p_4$ ,  $p_2 \leftrightarrow p_3$ , and  $p_1 \leftrightarrow p_3$ .
37. Show that the propositions  $p_1$ ,  $p_2$ ,  $p_3$ ,  $p_4$ , and  $p_5$  can be shown to be equivalent by proving that the conditional statements  $p_1 \rightarrow p_4$ ,  $p_3 \rightarrow p_1$ ,  $p_4 \rightarrow p_2$ ,  $p_2 \rightarrow p_5$ , and  $p_5 \rightarrow p_3$  are true.

# **Chapter (2)**

## **Propositional Logic**

# 1

# The Foundations: Logic and Proofs

- 1.1** Propositional Logic
- 1.2** Applications of Propositional Logic
- 1.3** Propositional Equivalences
- 1.4** Predicates and Quantifiers
- 1.5** Nested Quantifiers
- 1.6** Rules of Inference
- 1.7** Introduction to Proofs
- 1.8** Proof Methods and Strategy

**T**he rules of logic specify the meaning of mathematical statements. For instance, these rules help us understand and reason with statements such as “There exists an integer that is not the sum of two squares” and “For every positive integer  $n$ , the sum of the positive integers not exceeding  $n$  is  $n(n + 1)/2$ .” Logic is the basis of all mathematical reasoning, and of all automated reasoning. It has practical applications to the design of computing machines, to the specification of systems, to artificial intelligence, to computer programming, to programming languages, and to other areas of computer science, as well as to many other fields of study.

To understand mathematics, we must understand what makes up a correct mathematical argument, that is, a proof. Once we prove a mathematical statement is true, we call it a theorem. A collection of theorems on a topic organize what we know about this topic. To learn a mathematical topic, a person needs to actively construct mathematical arguments on this topic, and not just read exposition. Moreover, knowing the proof of a theorem often makes it possible to modify the result to fit new situations.

Everyone knows that proofs are important throughout mathematics, but many people find it surprising how important proofs are in computer science. In fact, proofs are used to verify that computer programs produce the correct output for all possible input values, to show that algorithms always produce the correct result, to establish the security of a system, and to create artificial intelligence. Furthermore, automated reasoning systems have been created to allow computers to construct their own proofs.

In this chapter, we will explain what makes up a correct mathematical argument and introduce tools to construct these arguments. We will develop an arsenal of different proof methods that will enable us to prove many different types of results. After introducing many different methods of proof, we will introduce several strategies for constructing proofs. We will introduce the notion of a conjecture and explain the process of developing mathematics by studying conjectures.

## 1.1 Propositional Logic

### Introduction

The rules of logic give precise meaning to mathematical statements. These rules are used to distinguish between valid and invalid mathematical arguments. Because a major goal of this book is to teach the reader how to understand and how to construct correct mathematical arguments, we begin our study of discrete mathematics with an introduction to logic.

Besides the importance of logic in understanding mathematical reasoning, logic has numerous applications to computer science. These rules are used in the design of computer circuits, the construction of computer programs, the verification of the correctness of programs, and in many other ways. Furthermore, software systems have been developed for constructing some, but not all, types of proofs automatically. We will discuss these applications of logic in this and later chapters.

## Propositions

Our discussion begins with an introduction to the basic building blocks of logic—propositions. A **proposition** is a declarative sentence (that is, a sentence that declares a fact) that is either true or false, but not both.

**EXAMPLE 1** All the following declarative sentences are propositions.



1. Washington, D.C., is the capital of the United States of America.
2. Toronto is the capital of Canada.
3.  $1 + 1 = 2$ .
4.  $2 + 2 = 3$ .

Propositions 1 and 3 are true, whereas 2 and 4 are false. 

Some sentences that are not propositions are given in Example 2.

**EXAMPLE 2** Consider the following sentences.

1. What time is it?
2. Read this carefully.
3.  $x + 1 = 2$ .
4.  $x + y = z$ .

Sentences 1 and 2 are not propositions because they are not declarative sentences. Sentences 3 and 4 are not propositions because they are neither true nor false. Note that each of sentences 3 and 4 can be turned into a proposition if we assign values to the variables. We will also discuss other ways to turn sentences such as these into propositions in Section 1.4. 

We use letters to denote **propositional variables** (or **statement variables**), that is, variables that represent propositions, just as letters are used to denote numerical variables. The



**ARISTOTLE (384 B.C.E.–322 B.C.E.)** Aristotle was born in Stagirus (Stagira) in northern Greece. His father was the personal physician of the King of Macedonia. Because his father died when Aristotle was young, Aristotle could not follow the custom of following his father's profession. Aristotle became an orphan at a young age when his mother also died. His guardian who raised him taught him poetry, rhetoric, and Greek. At the age of 17, his guardian sent him to Athens to further his education. Aristotle joined Plato's Academy, where for 20 years he attended Plato's lectures, later presenting his own lectures on rhetoric. When Plato died in 347 B.C.E., Aristotle was not chosen to succeed him because his views differed too much from those of Plato. Instead, Aristotle joined the court of King Hermeas where he remained for three years, and married the niece of King Philip of Macedonia. When the Persians defeated Hermeas, Aristotle moved to Mytilene and, at the invitation of King Philip of Macedonia, he tutored Alexander, Philip's son, who later became Alexander the Great. Aristotle tutored Alexander for five years and after the death of King Philip, he returned to Athens and set up his own school, called the Lyceum.

Aristotle's followers were called the peripatetics, which means "to walk about," because Aristotle often walked around as he discussed philosophical questions. Aristotle taught at the Lyceum for 13 years where he lectured to his advanced students in the morning and gave popular lectures to a broad audience in the evening. When Alexander the Great died in 323 B.C.E., a backlash against anything related to Alexander led to trumped-up charges of impiety against Aristotle. Aristotle fled to Chalcis to avoid prosecution. He only lived one year in Chalcis, dying of a stomach ailment in 322 B.C.E.

Aristotle wrote three types of works: those written for a popular audience, compilations of scientific facts, and systematic treatises. The systematic treatises included works on logic, philosophy, psychology, physics, and natural history. Aristotle's writings were preserved by a student and were hidden in a vault where a wealthy book collector discovered them about 200 years later. They were taken to Rome, where they were studied by scholars and issued in new editions, preserving them for posterity.

conventional letters used for propositional variables are  $p, q, r, s, \dots$ . The **truth value** of a proposition is true, denoted by T, if it is a true proposition, and the truth value of a proposition is false, denoted by F, if it is a false proposition.

The area of logic that deals with propositions is called the **propositional calculus** or **propositional logic**. It was first developed systematically by the Greek philosopher Aristotle more than 2300 years ago.



We now turn our attention to methods for producing new propositions from those that we already have. These methods were discussed by the English mathematician George Boole in 1854 in his book *The Laws of Thought*. Many mathematical statements are constructed by combining one or more propositions. New propositions, called **compound propositions**, are formed from existing propositions using logical operators.

### DEFINITION 1

Let  $p$  be a proposition. The *negation of  $p$* , denoted by  $\neg p$  (also denoted by  $\overline{p}$ ), is the statement

"It is not the case that  $p$ ."

The proposition  $\neg p$  is read "not  $p$ ." The truth value of the negation of  $p$ ,  $\neg p$ , is the opposite of the truth value of  $p$ .

### EXAMPLE 3

Find the negation of the proposition



"Michael's PC runs Linux"

and express this in simple English.

*Solution:* The negation is

"It is not the case that Michael's PC runs Linux."

This negation can be more simply expressed as

"Michael's PC does not run Linux."

### EXAMPLE 4

Find the negation of the proposition

"Vandana's smartphone has at least 32GB of memory"

and express this in simple English.

*Solution:* The negation is

"It is not the case that Vandana's smartphone has at least 32GB of memory."

This negation can also be expressed as

"Vandana's smartphone does not have at least 32GB of memory"

or even more simply as

"Vandana's smartphone has less than 32GB of memory."

**TABLE 1** The Truth Table for the Negation of a Proposition.

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

Table 1 displays the **truth table** for the negation of a proposition  $p$ . This table has a row for each of the two possible truth values of a proposition  $p$ . Each row shows the truth value of  $\neg p$  corresponding to the truth value of  $p$  for this row.

The negation of a proposition can also be considered the result of the operation of the **negation operator** on a proposition. The negation operator constructs a new proposition from a single existing proposition. We will now introduce the logical operators that are used to form new propositions from two or more existing propositions. These logical operators are also called **connectives**.

**DEFINITION 2**

Let  $p$  and  $q$  be propositions. The *conjunction* of  $p$  and  $q$ , denoted by  $p \wedge q$ , is the proposition “ $p$  and  $q$ .” The conjunction  $p \wedge q$  is true when both  $p$  and  $q$  are true and is false otherwise.

Table 2 displays the truth table of  $p \wedge q$ . This table has a row for each of the four possible combinations of truth values of  $p$  and  $q$ . The four rows correspond to the pairs of truth values TT, TF, FT, and FF, where the first truth value in the pair is the truth value of  $p$  and the second truth value is the truth value of  $q$ .

Note that in logic the word “but” sometimes is used instead of “and” in a conjunction. For example, the statement “The sun is shining, but it is raining” is another way of saying “The sun is shining and it is raining.” (In natural language, there is a subtle difference in meaning between “and” and “but”; we will not be concerned with this nuance here.)

**EXAMPLE 5**

Find the conjunction of the propositions  $p$  and  $q$  where  $p$  is the proposition “Rebecca’s PC has more than 16 GB free hard disk space” and  $q$  is the proposition “The processor in Rebecca’s PC runs faster than 1 GHz.”

**Solution:** The conjunction of these propositions,  $p \wedge q$ , is the proposition “Rebecca’s PC has more than 16 GB free hard disk space, and the processor in Rebecca’s PC runs faster than 1 GHz.” This conjunction can be expressed more simply as “Rebecca’s PC has more than 16 GB free hard disk space, and its processor runs faster than 1 GHz.” For this conjunction to be true, both conditions given must be true. It is false, when one or both of these conditions are false.  $\blacktriangleleft$

**DEFINITION 3**

Let  $p$  and  $q$  be propositions. The *disjunction* of  $p$  and  $q$ , denoted by  $p \vee q$ , is the proposition “ $p$  or  $q$ .” The disjunction  $p \vee q$  is false when both  $p$  and  $q$  are false and is true otherwise.

Table 3 displays the truth table for  $p \vee q$ .

**TABLE 2** The Truth Table for the Conjunction of Two Propositions.

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

**TABLE 3** The Truth Table for the Disjunction of Two Propositions.

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

The use of the connective *or* in a disjunction corresponds to one of the two ways the word *or* is used in English, namely, as an **inclusive or**. A disjunction is true when at least one of the two propositions is true. For instance, the inclusive or is being used in the statement

“Students who have taken calculus or computer science can take this class.”

Here, we mean that students who have taken both calculus and computer science can take the class, as well as the students who have taken only one of the two subjects. On the other hand, we are using the **exclusive or** when we say

“Students who have taken calculus or computer science, but not both, can enroll in this class.”

Here, we mean that students who have taken both calculus and a computer science course cannot take the class. Only those who have taken exactly one of the two courses can take the class.

Similarly, when a menu at a restaurant states, “Soup or salad comes with an entrée,” the restaurant almost always means that customers can have either soup or salad, but not both. Hence, this is an exclusive, rather than an inclusive, or.

**EXAMPLE 6** What is the disjunction of the propositions  $p$  and  $q$  where  $p$  and  $q$  are the same propositions as in Example 5?



*Solution:* The disjunction of  $p$  and  $q$ ,  $p \vee q$ , is the proposition

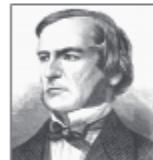
“Rebecca’s PC has at least 16 GB free hard disk space, or the processor in Rebecca’s PC runs faster than 1 GHz.”

This proposition is true when Rebecca’s PC has at least 16 GB free hard disk space, when the PC’s processor runs faster than 1 GHz, and when both conditions are true. It is false when both of these conditions are false, that is, when Rebecca’s PC has less than 16 GB free hard disk space and the processor in her PC runs at 1 GHz or slower. ▲

As was previously remarked, the use of the connective *or* in a disjunction corresponds to one of the two ways the word *or* is used in English, namely, in an inclusive way. Thus, a disjunction is true when at least one of the two propositions in it is true. Sometimes, we use *or* in an exclusive sense. When the exclusive or is used to connect the propositions  $p$  and  $q$ , the proposition “ $p$  or  $q$  (but not both)” is obtained. This proposition is true when  $p$  is true and  $q$  is false, and when  $p$  is false and  $q$  is true. It is false when both  $p$  and  $q$  are false and when both are true.



**GEORGE BOOLE (1815–1864)** George Boole, the son of a cobbler, was born in Lincoln, England, in November 1815. Because of his family’s difficult financial situation, Boole struggled to educate himself while supporting his family. Nevertheless, he became one of the most important mathematicians of the 1800s. Although he considered a career as a clergyman, he decided instead to go into teaching, and soon afterward opened a school of his own. In his preparation for teaching mathematics, Boole—unsatisfied with textbooks of his day—decided to read the works of the great mathematicians. While reading papers of the great French mathematician Lagrange, Boole made discoveries in the calculus of variations, the branch of analysis dealing with finding curves and surfaces by optimizing certain parameters.



In 1848 Boole published *The Mathematical Analysis of Logic*, the first of his contributions to symbolic logic. In 1849 he was appointed professor of mathematics at Queen’s College in Cork, Ireland. In 1854 he published *The Laws of Thought*, his most famous work. In this book, Boole introduced what is now called *Boolean algebra* in his honor. Boole wrote textbooks on differential equations and on difference equations that were used in Great Britain until the end of the nineteenth century. Boole married in 1855; his wife was the niece of the professor of Greek at Queen’s College. In 1864 Boole died from pneumonia, which he contracted as a result of keeping a lecture engagement even though he was soaking wet from a rainstorm.

**TABLE 4** The Truth Table for the Exclusive Or of Two Propositions.

$p$	$q$	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

**TABLE 5** The Truth Table for the Conditional Statement  $p \rightarrow q$ .

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

**DEFINITION 4**

Let  $p$  and  $q$  be propositions. The *exclusive or* of  $p$  and  $q$ , denoted by  $p \oplus q$ , is the proposition that is true when exactly one of  $p$  and  $q$  is true and is false otherwise.

The truth table for the exclusive or of two propositions is displayed in Table 4.

## Conditional Statements

We will discuss several other important ways in which propositions can be combined.

**DEFINITION 5**

Let  $p$  and  $q$  be propositions. The *conditional statement*  $p \rightarrow q$  is the proposition “if  $p$ , then  $q$ .” The conditional statement  $p \rightarrow q$  is false when  $p$  is true and  $q$  is false, and true otherwise. In the conditional statement  $p \rightarrow q$ ,  $p$  is called the *hypothesis* (or *antecedent* or *premise*) and  $q$  is called the *conclusion* (or *consequence*).



The statement  $p \rightarrow q$  is called a conditional statement because  $p \rightarrow q$  asserts that  $q$  is true on the condition that  $p$  holds. A conditional statement is also called an **implication**.

The truth table for the conditional statement  $p \rightarrow q$  is shown in Table 5. Note that the statement  $p \rightarrow q$  is true when both  $p$  and  $q$  are true and when  $p$  is false (no matter what truth value  $q$  has).

Because conditional statements play such an essential role in mathematical reasoning, a variety of terminology is used to express  $p \rightarrow q$ . You will encounter most if not all of the following ways to express this conditional statement:

“if $p$ , then $q$ ”	“ $p$ implies $q$ ”
“if $p, q$ ”	“ $p$ only if $q$ ”
“ $p$ is sufficient for $q$ ”	“a sufficient condition for $q$ is $p$ ”
“ $q$ if $p$ ”	“ $q$ whenever $p$ ”
“ $q$ when $p$ ”	“ $q$ is necessary for $p$ ”
“a necessary condition for $p$ is $q$ ”	“ $q$ follows from $p$ ”
“ $q$ unless $\neg p$ ”	

A useful way to understand the truth value of a conditional statement is to think of an obligation or a contract. For example, the pledge many politicians make when running for office is

“If I am elected, then I will lower taxes.”

If the politician is elected, voters would expect this politician to lower taxes. Furthermore, if the politician is not elected, then voters will not have any expectation that this person will lower taxes, although the person may have sufficient influence to cause those in power to lower taxes. It is only when the politician is elected but does not lower taxes that voters can say that the politician has broken the campaign pledge. This last scenario corresponds to the case when  $p$  is true but  $q$  is false in  $p \rightarrow q$ .

Similarly, consider a statement that a professor might make:

“If you get 100% on the final, then you will get an A.”

If you manage to get a 100% on the final, then you would expect to receive an A. If you do not get 100% you may or may not receive an A depending on other factors. However, if you do get 100%, but the professor does not give you an A, you will feel cheated.

Of the various ways to express the conditional statement  $p \rightarrow q$ , the two that seem to cause the most confusion are “ $p$  only if  $q$ ” and “ $q$  unless  $\neg p$ .” Consequently, we will provide some guidance for clearing up this confusion.

To remember that “ $p$  only if  $q$ ” expresses the same thing as “if  $p$ , then  $q$ ,” note that “ $p$  only if  $q$ ” says that  $p$  cannot be true when  $q$  is not true. That is, the statement is false if  $p$  is true, but  $q$  is false. When  $p$  is false,  $q$  may be either true or false, because the statement says nothing about the truth value of  $q$ . Be careful not to use “ $q$  only if  $p$ ” to express  $p \rightarrow q$  because this is incorrect. To see this, note that the true values of “ $q$  only if  $p$ ” and  $p \rightarrow q$  are different when  $p$  and  $q$  have different truth values.

To remember that “ $q$  unless  $\neg p$ ” expresses the same conditional statement as “if  $p$ , then  $q$ ,” note that “ $q$  unless  $\neg p$ ” means that if  $\neg p$  is false, then  $q$  must be true. That is, the statement “ $q$  unless  $\neg p$ ” is false when  $p$  is true but  $q$  is false, but it is true otherwise. Consequently, “ $q$  unless  $\neg p$ ” and  $p \rightarrow q$  always have the same truth value.

We illustrate the translation between conditional statements and English statements in Example 7.

### EXAMPLE 7

Let  $p$  be the statement “Maria learns discrete mathematics” and  $q$  the statement “Maria will find a good job.” Express the statement  $p \rightarrow q$  as a statement in English.



**Solution:** From the definition of conditional statements, we see that when  $p$  is the statement “Maria learns discrete mathematics” and  $q$  is the statement “Maria will find a good job,”  $p \rightarrow q$  represents the statement

“If Maria learns discrete mathematics, then she will find a good job.”

There are many other ways to express this conditional statement in English. Among the most natural of these are:

“Maria will find a good job when she learns discrete mathematics.”

“For Maria to get a good job, it is sufficient for her to learn discrete mathematics.”

and

“Maria will find a good job unless she does not learn discrete mathematics.”

Note that the way we have defined conditional statements is more general than the meaning attached to such statements in the English language. For instance, the conditional statement in Example 7 and the statement

“If it is sunny, then we will go to the beach.”

are statements used in normal language where there is a relationship between the hypothesis and the conclusion. Further, the first of these statements is true unless Maria learns discrete mathematics, but she does not get a good job, and the second is true unless it is indeed sunny, but we do not go to the beach. On the other hand, the statement

"If Juan has a smartphone, then  $2 + 3 = 5$ "

is true from the definition of a conditional statement, because its conclusion is true. (The truth value of the hypothesis does not matter then.) The conditional statement

"If Juan has a smartphone, then  $2 + 3 = 6$ "

is true if Juan does not have a smartphone, even though  $2 + 3 = 6$  is false. We would not use these last two conditional statements in natural language (except perhaps in sarcasm), because there is no relationship between the hypothesis and the conclusion in either statement. In mathematical reasoning, we consider conditional statements of a more general sort than we use in English. The mathematical concept of a conditional statement is independent of a cause-and-effect relationship between hypothesis and conclusion. Our definition of a conditional statement specifies its truth values; it is not based on English usage. Propositional language is an artificial language; we only parallel English usage to make it easy to use and remember.

The if-then construction used in many programming languages is different from that used in logic. Most programming languages contain statements such as **if**  $p$  **then**  $S$ , where  $p$  is a proposition and  $S$  is a program segment (one or more statements to be executed). When execution of a program encounters such a statement,  $S$  is executed if  $p$  is true, but  $S$  is not executed if  $p$  is false, as illustrated in Example 8.

**EXAMPLE 8** What is the value of the variable  $x$  after the statement

**if**  $2 + 2 = 4$  **then**  $x := x + 1$

if  $x = 0$  before this statement is encountered? (The symbol  $:=$  stands for assignment. The statement  $x := x + 1$  means the assignment of the value of  $x + 1$  to  $x$ .)

**Solution:** Because  $2 + 2 = 4$  is true, the assignment statement  $x := x + 1$  is executed. Hence,  $x$  has the value  $0 + 1 = 1$  after this statement is encountered. 

**CONVERSE, CONTRAPOSITIVE, AND INVERSE** We can form some new conditional statements starting with a conditional statement  $p \rightarrow q$ . In particular, there are three related conditional statements that occur so often that they have special names. The proposition  $q \rightarrow p$  is called the **converse** of  $p \rightarrow q$ . The **contrapositive** of  $p \rightarrow q$  is the proposition  $\neg q \rightarrow \neg p$ . The proposition  $\neg p \rightarrow \neg q$  is called the **inverse** of  $p \rightarrow q$ . We will see that of these three conditional statements formed from  $p \rightarrow q$ , only the contrapositive always has the same truth value as  $p \rightarrow q$ .

We first show that the contrapositive,  $\neg q \rightarrow \neg p$ , of a conditional statement  $p \rightarrow q$  always has the same truth value as  $p \rightarrow q$ . To see this, note that the contrapositive is false only when  $\neg p$  is false and  $\neg q$  is true, that is, only when  $p$  is true and  $q$  is false. We now show that neither the converse,  $q \rightarrow p$ , nor the inverse,  $\neg p \rightarrow \neg q$ , has the same truth value as  $p \rightarrow q$  for all possible truth values of  $p$  and  $q$ . Note that when  $p$  is true and  $q$  is false, the original conditional statement is false, but the converse and the inverse are both true.

When two compound propositions always have the same truth value we call them **equivalent**, so that a conditional statement and its contrapositive are equivalent. The converse and the inverse of a conditional statement are also equivalent, as the reader can verify, but neither is equivalent to the original conditional statement. (We will study equivalent propositions in Section 1.3.) Take note that one of the most common logical errors is to assume that the converse or the inverse of a conditional statement is equivalent to this conditional statement.

We illustrate the use of conditional statements in Example 9.

Remember that the contrapositive, but neither the converse or inverse, of a conditional statement is equivalent to it.

**EXAMPLE 9** What are the contrapositive, the converse, and the inverse of the conditional statement  
“The home team wins whenever it is raining?”



**Solution:** Because “ $q$  whenever  $p$ ” is one of the ways to express the conditional statement  $p \rightarrow q$ , the original statement can be rewritten as

“If it is raining, then the home team wins.”

Consequently, the contrapositive of this conditional statement is

“If the home team does not win, then it is not raining.”

The converse is

“If the home team wins, then it is raining.”

The inverse is

“If it is not raining, then the home team does not win.”

Only the contrapositive is equivalent to the original statement.

**BICONDITIONALS** We now introduce another way to combine propositions that expresses that two propositions have the same truth value.

### DEFINITION 6

Let  $p$  and  $q$  be propositions. The *biconditional statement*  $p \leftrightarrow q$  is the proposition “ $p$  if and only if  $q$ .” The biconditional statement  $p \leftrightarrow q$  is true when  $p$  and  $q$  have the same truth values, and is false otherwise. Biconditional statements are also called *bi-implications*.

The truth table for  $p \leftrightarrow q$  is shown in Table 6. Note that the statement  $p \leftrightarrow q$  is true when both the conditional statements  $p \rightarrow q$  and  $q \rightarrow p$  are true and is false otherwise. That is why we use the words “if and only if” to express this logical connective and why it is symbolically written by combining the symbols  $\rightarrow$  and  $\leftarrow$ . There are some other common ways to express  $p \leftrightarrow q$ :

- “ $p$  is necessary and sufficient for  $q$ ”
- “if  $p$  then  $q$ , and conversely”
- “ $p$  iff  $q$ .”

The last way of expressing the biconditional statement  $p \leftrightarrow q$  uses the abbreviation “iff” for “if and only if.” Note that  $p \leftrightarrow q$  has exactly the same truth value as  $(p \rightarrow q) \wedge (q \rightarrow p)$ .

**TABLE 6** The Truth Table for the Biconditional  $p \leftrightarrow q$ .

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

**EXAMPLE 10** Let  $p$  be the statement “You can take the flight,” and let  $q$  be the statement “You buy a ticket.” Then  $p \leftrightarrow q$  is the statement

“You can take the flight if and only if you buy a ticket.”



This statement is true if  $p$  and  $q$  are either both true or both false, that is, if you buy a ticket and can take the flight or if you do not buy a ticket and you cannot take the flight. It is false when  $p$  and  $q$  have opposite truth values, that is, when you do not buy a ticket, but you can take the flight (such as when you get a free trip) and when you buy a ticket but you cannot take the flight (such as when the airline bumps you).

**IMPLICIT USE OF BICONDITIONALS** You should be aware that biconditionals are not always explicit in natural language. In particular, the “if and only if” construction used in biconditionals is rarely used in common language. Instead, biconditionals are often expressed using an “if, then” or an “only if” construction. The other part of the “if and only if” is implicit. That is, the converse is implied, but not stated. For example, consider the statement in English “If you finish your meal, then you can have dessert.” What is really meant is “You can have dessert if and only if you finish your meal.” This last statement is logically equivalent to the two statements “If you finish your meal, then you can have dessert” and “You can have dessert only if you finish your meal.” Because of this imprecision in natural language, we need to make an assumption whether a conditional statement in natural language implicitly includes its converse. Because precision is essential in mathematics and in logic, we will always distinguish between the conditional statement  $p \rightarrow q$  and the biconditional statement  $p \leftrightarrow q$ .

## Truth Tables of Compound Propositions



We have now introduced four important logical connectives—conjunctions, disjunctions, conditional statements, and biconditional statements—as well as negations. We can use these connectives to build up complicated compound propositions involving any number of propositional variables. We can use truth tables to determine the truth values of these compound propositions, as Example 11 illustrates. We use a separate column to find the truth value of each compound expression that occurs in the compound proposition as it is built up. The truth values of the compound proposition for each combination of truth values of the propositional variables in it is found in the final column of the table.

**EXAMPLE 11** Construct the truth table of the compound proposition

$$(p \vee \neg q) \rightarrow (p \wedge q).$$

**Solution:** Because this truth table involves two propositional variables  $p$  and  $q$ , there are four rows in this truth table, one for each of the pairs of truth values TT, TF, FT, and FF. The first two columns are used for the truth values of  $p$  and  $q$ , respectively. In the third column we find the truth value of  $\neg q$ , needed to find the truth value of  $p \vee \neg q$ , found in the fourth column. The fifth column gives the truth value of  $p \wedge q$ . Finally, the truth value of  $(p \vee \neg q) \rightarrow (p \wedge q)$  is found in the last column. The resulting truth table is shown in Table 7.

**TABLE 7** The Truth Table of  $(p \vee \neg q) \rightarrow (p \wedge q)$ .

$p$	$q$	$\neg q$	$p \vee \neg q$	$p \wedge q$	$(p \vee \neg q) \rightarrow (p \wedge q)$
T	T	F	T	T	T
T	F	T	T	F	F
F	T	F	F	F	T
F	F	T	T	F	F

## Precedence of Logical Operators

**TABLE 8**  
Precedence of  
Logical Operators.

Operator	Precedence
$\neg$	1
$\wedge$	2
$\vee$	3
$\rightarrow$	4
$\leftrightarrow$	5

We can construct compound propositions using the negation operator and the logical operators defined so far. We will generally use parentheses to specify the order in which logical operators in a compound proposition are to be applied. For instance,  $(p \vee q) \wedge (\neg r)$  is the conjunction of  $p \vee q$  and  $\neg r$ . However, to reduce the number of parentheses, we specify that the negation operator is applied before all other logical operators. This means that  $\neg p \wedge q$  is the conjunction of  $\neg p$  and  $q$ , namely,  $(\neg p) \wedge q$ , not the negation of the conjunction of  $p$  and  $q$ , namely  $\neg(p \wedge q)$ .

Another general rule of precedence is that the conjunction operator takes precedence over the disjunction operator, so that  $p \wedge q \vee r$  means  $(p \wedge q) \vee r$  rather than  $p \wedge (q \vee r)$ . Because this rule may be difficult to remember, we will continue to use parentheses so that the order of the disjunction and conjunction operators is clear.

Finally, it is an accepted rule that the conditional and biconditional operators  $\rightarrow$  and  $\leftrightarrow$  have lower precedence than the conjunction and disjunction operators,  $\wedge$  and  $\vee$ . Consequently,  $p \vee q \rightarrow r$  is the same as  $(p \vee q) \rightarrow r$ . We will use parentheses when the order of the conditional operator and biconditional operator is at issue, although the conditional operator has precedence over the biconditional operator. Table 8 displays the precedence levels of the logical operators,  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\rightarrow$ , and  $\leftrightarrow$ .

## Logic and Bit Operations

Computers represent information using bits. A **bit** is a symbol with two possible values, namely, 0 (zero) and 1 (one). This meaning of the word bit comes from *binary digit*, because zeros and ones are the digits used in binary representations of numbers. The well-known statistician John Tukey introduced this terminology in 1946. A bit can be used to represent a truth value, because there are two truth values, namely, *true* and *false*. As is customarily done, we will use a 1 bit to represent true and a 0 bit to represent false. That is, 1 represents T (true), 0 represents F (false). A variable is called a **Boolean variable** if its value is either true or false. Consequently, a Boolean variable can be represented using a bit.

Computer **bit operations** correspond to the logical connectives. By replacing true by a one and false by a zero in the truth tables for the operators  $\wedge$ ,  $\vee$ , and  $\oplus$ , the tables shown in Table 9 for the corresponding bit operations are obtained. We will also use the notation *OR*, *AND*, and *XOR* for the operators  $\vee$ ,  $\wedge$ , and  $\oplus$ , as is done in various programming languages.



**JOHN WILDER TUKEY (1915–2000)** Tukey, born in New Bedford, Massachusetts, was an only child. His parents, both teachers, decided home schooling would best develop his potential. His formal education began at Brown University, where he studied mathematics and chemistry. He received a master's degree in chemistry from Brown and continued his studies at Princeton University, changing his field of study from chemistry to mathematics. He received his Ph.D. from Princeton in 1939 for work in topology, when he was appointed an instructor in mathematics at Princeton. With the start of World War II, he joined the Fire Control Research Office, where he began working in statistics. Tukey found statistical research to his liking and impressed several leading statisticians with his skills. In 1945, at the conclusion of the war, Tukey returned to the mathematics department at Princeton as a professor of statistics, and he also took a position at AT&T Bell Laboratories. Tukey founded the Statistics Department at Princeton in 1966 and was its first chairman. Tukey made significant contributions to many areas of statistics, including the analysis of variance, the estimation of spectra of time series, inferences about the values of a set of parameters from a single experiment, and the philosophy of statistics. However, he is best known for his invention, with J. W. Cooley, of the fast Fourier transform. In addition to his contributions to statistics, Tukey was noted as a skilled wordsmith; he is credited with coining the terms *bit* and *software*.

Tukey contributed his insight and expertise by serving on the President's Science Advisory Committee. He chaired several important committees dealing with the environment, education, and chemicals and health. He also served on committees working on nuclear disarmament. Tukey received many awards, including the National Medal of Science.

**HISTORICAL NOTE** There were several other suggested words for a binary digit, including *binit* and *bigit*, that never were widely accepted. The adoption of the word *bit* may be due to its meaning as a common English word. For an account of Tukey's coining of the word *bit*, see the April 1984 issue of *Annals of the History of Computing*.

**TABLE 9** Table for the Bit Operators *OR*, *AND*, and *XOR*.

$x$	$y$	$x \vee y$	$x \wedge y$	$x \oplus y$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Information is often represented using bit strings, which are lists of zeros and ones. When this is done, operations on the bit strings can be used to manipulate this information.

### DEFINITION 7

A *bit string* is a sequence of zero or more bits. The *length* of this string is the number of bits in the string.

**EXAMPLE 12** 101010011 is a bit string of length nine.

We can extend bit operations to bit strings. We define the **bitwise OR**, **bitwise AND**, and **bitwise XOR** of two strings of the same length to be the strings that have as their bits the *OR*, *AND*, and *XOR* of the corresponding bits in the two strings, respectively. We use the symbols  $\vee$ ,  $\wedge$ , and  $\oplus$  to represent the bitwise *OR*, bitwise *AND*, and bitwise *XOR* operations, respectively. We illustrate bitwise operations on bit strings with Example 13.

**EXAMPLE 13** Find the bitwise *OR*, bitwise *AND*, and bitwise *XOR* of the bit strings 0110110110 and 1100011101. (Here, and throughout this book, bit strings will be split into blocks of four bits to make them easier to read.)

**Solution:** The bitwise *OR*, bitwise *AND*, and bitwise *XOR* of these strings are obtained by taking the *OR*, *AND*, and *XOR* of the corresponding bits, respectively. This gives us

$$\begin{array}{r} 01\ 1011\ 0110 \\ 11\ 0001\ 1101 \\ \hline 11\ 1011\ 1111 & \text{bitwise OR} \\ 01\ 0001\ 0100 & \text{bitwise AND} \\ 10\ 1010\ 1011 & \text{bitwise XOR} \end{array}$$

## Exercises

- Which of these sentences are propositions? What are the truth values of those that are propositions?
  - Boston is the capital of Massachusetts.
  - Miami is the capital of Florida.
  - $2 + 3 = 5$ .
  - $5 + 7 = 10$ .
  - $x + 2 = 11$ .
  - Answer this question.
- Which of these are propositions? What are the truth values of those that are propositions?
  - Do not pass go.
  - What time is it?
  - There are no black flies in Maine.
- What is the negation of each of these propositions?
  - Mei has an MP3 player.
  - There is no pollution in New Jersey.
  - $2 + 1 = 3$ .
  - The summer in Maine is hot and sunny.
- What is the negation of each of these propositions?
  - Jennifer and Teja are friends.
  - There are 13 items in a baker's dozen.
  - Abby sent more than 100 text messages every day.
  - 121 is a perfect square.

5. What is the negation of each of these propositions?
- Steve has more than 100 GB free disk space on his laptop.
  - Zach blocks e-mails and texts from Jennifer.
  - $7 \cdot 11 \cdot 13 = 999$ .
  - Diane rode her bicycle 100 miles on Sunday.
6. Suppose that Smartphone A has 256 MB RAM and 32 GB ROM, and the resolution of its camera is 8 MP; Smartphone B has 288 MB RAM and 64 GB ROM, and the resolution of its camera is 4 MP; and Smartphone C has 128 MB RAM and 32 GB ROM, and the resolution of its camera is 5 MP. Determine the truth value of each of these propositions.
- Smartphone B has the most RAM of these three smartphones.
  - Smartphone C has more ROM or a higher resolution camera than Smartphone B.
  - Smartphone B has more RAM, more ROM, and a higher resolution camera than Smartphone A.
  - If Smartphone B has more RAM and more ROM than Smartphone C, then it also has a higher resolution camera.
  - Smartphone A has more RAM than Smartphone B if and only if Smartphone B has more RAM than Smartphone A.
7. Suppose that during the most recent fiscal year, the annual revenue of Acme Computer was 138 billion dollars and its net profit was 8 billion dollars, the annual revenue of Nadir Software was 87 billion dollars and its net profit was 5 billion dollars, and the annual revenue of Quixote Media was 111 billion dollars and its net profit was 13 billion dollars. Determine the truth value of each of these propositions for the most recent fiscal year.
- Quixote Media had the largest annual revenue.
  - Nadir Software had the lowest net profit and Acme Computer had the largest annual revenue.
  - Acme Computer had the largest net profit or Quixote Media had the largest net profit.
  - If Quixote Media had the smallest net profit, then Acme Computer had the largest annual revenue.
  - Nadir Software had the smallest net profit if and only if Acme Computer had the largest annual revenue.
8. Let  $p$  and  $q$  be the propositions
- $p$  : I bought a lottery ticket this week.  
 $q$  : I won the million dollar jackpot.
- Express each of these propositions as an English sentence.
- $\neg p$
  - $p \vee q$
  - $p \rightarrow q$
  - $p \wedge q$
  - $p \leftrightarrow q$
  - $\neg p \rightarrow \neg q$
  - $\neg p \wedge \neg q$
  - $\neg p \vee (p \wedge q)$
9. Let  $p$  and  $q$  be the propositions “Swimming at the New Jersey shore is allowed” and “Sharks have been spotted near the shore,” respectively. Express each of these compound propositions as an English sentence.
- $\neg q$
  - $p \wedge q$
  - $\neg p \vee q$
  - $p \rightarrow \neg q$
  - $\neg q \rightarrow p$
  - $\neg p \rightarrow \neg q$
  - $p \leftrightarrow \neg q$
  - $\neg p \wedge (p \vee \neg q)$

10. Let  $p$  and  $q$  be the propositions “The election is decided” and “The votes have been counted,” respectively. Express each of these compound propositions as an English sentence.
- $\neg p$
  - $p \vee q$
  - $\neg p \wedge q$
  - $q \rightarrow p$
  - $\neg q \rightarrow \neg p$
  - $\neg p \rightarrow \neg q$
  - $p \leftrightarrow q$
  - $\neg q \vee (\neg p \wedge q)$
11. Let  $p$  and  $q$  be the propositions
- $p$  : It is below freezing.  
 $q$  : It is snowing.
- Write these propositions using  $p$  and  $q$  and logical connectives (including negations).
- It is below freezing and snowing.
  - It is below freezing but not snowing.
  - It is not below freezing and it is not snowing.
  - It is either snowing or below freezing (or both).
  - If it is below freezing, it is also snowing.
  - Either it is below freezing or it is snowing, but it is not snowing if it is below freezing.
  - That it is below freezing is necessary and sufficient for it to be snowing.
12. Let  $p$ ,  $q$ , and  $r$  be the propositions
- $p$  : You have the flu.  
 $q$  : You miss the final examination.  
 $r$  : You pass the course.
- Express each of these propositions as an English sentence.
- $p \rightarrow q$
  - $\neg q \leftrightarrow r$
  - $q \rightarrow \neg r$
  - $p \vee q \vee r$
  - $(p \rightarrow \neg r) \vee (q \rightarrow \neg r)$
  - $(p \wedge q) \vee (\neg q \wedge r)$
13. Let  $p$  and  $q$  be the propositions
- $p$  : You drive over 65 miles per hour.  
 $q$  : You get a speeding ticket.
- Write these propositions using  $p$  and  $q$  and logical connectives (including negations).
- You do not drive over 65 miles per hour.
  - You drive over 65 miles per hour, but you do not get a speeding ticket.
  - You will get a speeding ticket if you drive over 65 miles per hour.
  - If you do not drive over 65 miles per hour, then you will not get a speeding ticket.
  - Driving over 65 miles per hour is sufficient for getting a speeding ticket.
  - You get a speeding ticket, but you do not drive over 65 miles per hour.
  - Whenever you get a speeding ticket, you are driving over 65 miles per hour.
14. Let  $p$ ,  $q$ , and  $r$  be the propositions
- $p$  : You get an A on the final exam.  
 $q$  : You do every exercise in this book.  
 $r$  : You get an A in this class.
- Write these propositions using  $p$ ,  $q$ , and  $r$  and logical connectives (including negations).

- a) You get an A in this class, but you do not do every exercise in this book.
- b) You get an A on the final, you do every exercise in this book, and you get an A in this class.
- c) To get an A in this class, it is necessary for you to get an A on the final.
- d) You get an A on the final, but you don't do every exercise in this book; nevertheless, you get an A in this class.
- e) Getting an A on the final and doing every exercise in this book is sufficient for getting an A in this class.
- f) You will get an A in this class if and only if you either do every exercise in this book or you get an A on the final.
15. Let  $p$ ,  $q$ , and  $r$  be the propositions
- $p$  : Grizzly bears have been seen in the area.
  - $q$  : Hiking is safe on the trail.
  - $r$  : Berries are ripe along the trail.
- Write these propositions using  $p$ ,  $q$ , and  $r$  and logical connectives (including negations).
- a) Berries are ripe along the trail, but grizzly bears have not been seen in the area.
- b) Grizzly bears have not been seen in the area and hiking on the trail is safe, but berries are ripe along the trail.
- c) If berries are ripe along the trail, hiking is safe if and only if grizzly bears have not been seen in the area.
- d) It is not safe to hike on the trail, but grizzly bears have not been seen in the area and the berries along the trail are ripe.
- e) For hiking on the trail to be safe, it is necessary but not sufficient that berries not be ripe along the trail and for grizzly bears not to have been seen in the area.
- f) Hiking is not safe on the trail whenever grizzly bears have been seen in the area and berries are ripe along the trail.
16. Determine whether these biconditionals are true or false.
- a)  $2 + 2 = 4$  if and only if  $1 + 1 = 2$ .
- b)  $1 + 1 = 2$  if and only if  $2 + 3 = 4$ .
- c)  $1 + 1 = 3$  if and only if monkeys can fly.
- d)  $0 > 1$  if and only if  $2 > 1$ .
17. Determine whether each of these conditional statements is true or false.
- a) If  $1 + 1 = 2$ , then  $2 + 2 = 5$ .
- b) If  $1 + 1 = 3$ , then  $2 + 2 = 4$ .
- c) If  $1 + 1 = 3$ , then  $2 + 2 = 5$ .
- d) If monkeys can fly, then  $1 + 1 = 3$ .
18. Determine whether each of these conditional statements is true or false.
- a) If  $1 + 1 = 3$ , then unicorns exist.
- b) If  $1 + 1 = 3$ , then dogs can fly.
- c) If  $1 + 1 = 2$ , then dogs can fly.
- d) If  $2 + 2 = 4$ , then  $1 + 2 = 3$ .
19. For each of these sentences, determine whether an inclusive or, or an exclusive or, is intended. Explain your answer.
- a) Coffee or tea comes with dinner.
- b) A password must have at least three digits or be at least eight characters long.
- c) The prerequisite for the course is a course in number theory or a course in cryptography.
- d) You can pay using U.S. dollars or euros.
20. For each of these sentences, determine whether an inclusive or, or an exclusive or, is intended. Explain your answer.
- a) Experience with C++ or Java is required.
- b) Lunch includes soup or salad.
- c) To enter the country you need a passport or a voter registration card.
- d) Publish or perish.
21. For each of these sentences, state what the sentence means if the logical connective or is an inclusive or (that is, a disjunction) versus an exclusive or. Which of these meanings of or do you think is intended?
- a) To take discrete mathematics, you must have taken calculus or a course in computer science.
- b) When you buy a new car from Acme Motor Company, you get \$2000 back in cash or a 2% car loan.
- c) Dinner for two includes two items from column A or three items from column B.
- d) School is closed if more than 2 feet of snow falls or if the wind chill is below  $-100$ .
22. Write each of these statements in the form "if  $p$ , then  $q$ " in English. [Hint: Refer to the list of common ways to express conditional statements provided in this section.]
- a) It is necessary to wash the boss's car to get promoted.
- b) Winds from the south imply a spring thaw.
- c) A sufficient condition for the warranty to be good is that you bought the computer less than a year ago.
- d) Willy gets caught whenever he cheats.
- e) You can access the website only if you pay a subscription fee.
- f) Getting elected follows from knowing the right people.
- g) Carol gets seasick whenever she is on a boat.
23. Write each of these statements in the form "if  $p$ , then  $q$ " in English. [Hint: Refer to the list of common ways to express conditional statements.]
- a) It snows whenever the wind blows from the northeast.
- b) The apple trees will bloom if it stays warm for a week.
- c) That the Pistons win the championship implies that they beat the Lakers.
- d) It is necessary to walk 8 miles to get to the top of Long's Peak.
- e) To get tenure as a professor, it is sufficient to be world-famous.
- f) If you drive more than 400 miles, you will need to buy gasoline.
- g) Your guarantee is good only if you bought your CD player less than 90 days ago.
- h) Jan will go swimming unless the water is too cold.

- 24.** Write each of these statements in the form “if  $p$ , then  $q$ ” in English. [Hint: Refer to the list of common ways to express conditional statements provided in this section.]
- I will remember to send you the address only if you send me an e-mail message.
  - To be a citizen of this country, it is sufficient that you were born in the United States.
  - If you keep your textbook, it will be a useful reference in your future courses.
  - The Red Wings will win the Stanley Cup if their goalie plays well.
  - That you get the job implies that you had the best credentials.
  - The beach erodes whenever there is a storm.
  - It is necessary to have a valid password to log on to the server.
  - You will reach the summit unless you begin your climb too late.
- 25.** Write each of these propositions in the form “ $p$  if and only if  $q$ ” in English.
- If it is hot outside you buy an ice cream cone, and if you buy an ice cream cone it is hot outside.
  - For you to win the contest it is necessary and sufficient that you have the only winning ticket.
  - You get promoted only if you have connections, and you have connections only if you get promoted.
  - If you watch television your mind will decay, and conversely.
  - The trains run late on exactly those days when I take it.
- 26.** Write each of these propositions in the form “ $p$  if and only if  $q$ ” in English.
- For you to get an A in this course, it is necessary and sufficient that you learn how to solve discrete mathematics problems.
  - If you read the newspaper every day, you will be informed, and conversely.
  - It rains if it is a weekend day, and it is a weekend day if it rains.
  - You can see the wizard only if the wizard is not in, and the wizard is not in only if you can see him.
- 27.** State the converse, contrapositive, and inverse of each of these conditional statements.
- If it snows today, I will ski tomorrow.
  - I come to class whenever there is going to be a quiz.
  - A positive integer is a prime only if it has no divisors other than 1 and itself.
- 28.** State the converse, contrapositive, and inverse of each of these conditional statements.
- If it snows tonight, then I will stay at home.
  - I go to the beach whenever it is a sunny summer day.
  - When I stay up late, it is necessary that I sleep until noon.
- 29.** How many rows appear in a truth table for each of these compound propositions?
- $p \rightarrow \neg p$
  - $(p \vee \neg r) \wedge (q \vee \neg s)$
  - $q \vee p \vee \neg s \vee \neg r \vee \neg t \vee u$
  - $(p \wedge r \wedge t) \leftrightarrow (q \wedge t)$
- 30.** How many rows appear in a truth table for each of these compound propositions?
- $(q \rightarrow \neg p) \vee (\neg p \rightarrow \neg q)$
  - $(p \vee \neg t) \wedge (p \vee \neg s)$
  - $(p \rightarrow r) \vee (\neg s \rightarrow \neg t) \vee (\neg u \rightarrow v)$
  - $(p \wedge r \wedge s) \vee (q \wedge t) \vee (r \wedge \neg t)$
- 31.** Construct a truth table for each of these compound propositions.
- $p \wedge \neg p$
  - $p \vee \neg p$
  - $(p \vee \neg q) \rightarrow q$
  - $(p \vee q) \rightarrow (p \wedge q)$
  - $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$
  - $(p \rightarrow q) \rightarrow (q \rightarrow p)$
- 32.** Construct a truth table for each of these compound propositions.
- $p \rightarrow \neg p$
  - $p \leftrightarrow \neg p$
  - $p \oplus (p \vee q)$
  - $(p \wedge q) \rightarrow (p \vee q)$
  - $(q \rightarrow \neg p) \leftrightarrow (p \leftrightarrow q)$
  - $(p \leftrightarrow q) \oplus (p \leftrightarrow \neg q)$
- 33.** Construct a truth table for each of these compound propositions.
- $(p \vee q) \rightarrow (p \oplus q)$
  - $(p \oplus q) \rightarrow (p \wedge q)$
  - $(p \vee q) \oplus (p \wedge q)$
  - $(p \leftrightarrow q) \oplus (\neg p \leftrightarrow \neg q)$
  - $(p \leftrightarrow q) \oplus (\neg p \leftrightarrow \neg r)$
  - $(p \oplus q) \rightarrow (p \oplus \neg q)$
- 34.** Construct a truth table for each of these compound propositions.
- $p \oplus p$
  - $p \oplus \neg p$
  - $p \oplus \neg q$
  - $\neg p \oplus \neg q$
  - $(p \oplus q) \vee (p \oplus \neg q)$
  - $(p \oplus q) \wedge (p \oplus \neg q)$
- 35.** Construct a truth table for each of these compound propositions.
- $p \rightarrow \neg q$
  - $\neg p \leftrightarrow q$
  - $(p \rightarrow q) \vee (\neg p \rightarrow q)$
  - $(p \rightarrow q) \wedge (\neg p \rightarrow q)$
  - $(p \leftrightarrow q) \vee (\neg p \leftrightarrow q)$
  - $(\neg p \leftrightarrow \neg q) \leftrightarrow (p \leftrightarrow q)$
- 36.** Construct a truth table for each of these compound propositions.
- $(p \vee q) \vee r$
  - $(p \vee q) \wedge r$
  - $(p \wedge q) \vee r$
  - $(p \wedge q) \wedge r$
  - $(p \vee q) \wedge \neg r$
  - $(p \wedge q) \vee \neg r$
- 37.** Construct a truth table for each of these compound propositions.
- $p \rightarrow (\neg q \vee r)$
  - $\neg p \rightarrow (q \rightarrow r)$
  - $(p \rightarrow q) \vee (\neg p \rightarrow r)$
  - $(p \rightarrow q) \wedge (\neg p \rightarrow r)$
  - $(p \leftrightarrow q) \vee (\neg q \leftrightarrow r)$
  - $(\neg p \leftrightarrow \neg q) \leftrightarrow (q \leftrightarrow r)$
- 38.** Construct a truth table for  $((p \rightarrow q) \rightarrow r) \rightarrow s$ .
- 39.** Construct a truth table for  $(p \leftrightarrow q) \leftrightarrow (r \leftrightarrow s)$ .

## 1.3 Propositional Equivalences

### Introduction

An important type of step used in a mathematical argument is the replacement of a statement with another statement with the same truth value. Because of this, methods that produce propositions with the same truth value as a given compound proposition are used extensively in the construction of mathematical arguments. Note that we will use the term “compound proposition” to refer to an expression formed from propositional variables using logical operators, such as  $p \wedge q$ .

We begin our discussion with a classification of compound propositions according to their possible truth values.

#### DEFINITION 1

A compound proposition that is always true, no matter what the truth values of the propositional variables that occur in it, is called a *tautology*. A compound proposition that is always false is called a *contradiction*. A compound proposition that is neither a tautology nor a contradiction is called a *contingency*.

Tautologies and contradictions are often important in mathematical reasoning. Example 1 illustrates these types of compound propositions.

#### EXAMPLE 1

We can construct examples of tautologies and contradictions using just one propositional variable. Consider the truth tables of  $p \vee \neg p$  and  $p \wedge \neg p$ , shown in Table 1. Because  $p \vee \neg p$  is always true, it is a tautology. Because  $p \wedge \neg p$  is always false, it is a contradiction. ▲



### Logical Equivalences

Compound propositions that have the same truth values in all possible cases are called **logically equivalent**. We can also define this notion as follows.

#### DEFINITION 2

The compound propositions  $p$  and  $q$  are called *logically equivalent* if  $p \leftrightarrow q$  is a tautology. The notation  $p \equiv q$  denotes that  $p$  and  $q$  are logically equivalent.

**Remark:** The symbol  $\equiv$  is not a logical connective, and  $p \equiv q$  is not a compound proposition but rather is the statement that  $p \leftrightarrow q$  is a tautology. The symbol  $\Leftrightarrow$  is sometimes used instead of  $\equiv$  to denote logical equivalence.

One way to determine whether two compound propositions are equivalent is to use a truth table. In particular, the compound propositions  $p$  and  $q$  are equivalent if and only if the columns

TABLE 1 Examples of a Tautology and a Contradiction.

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

**TABLE 2** De Morgan's Laws.

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$



giving their truth values agree. Example 2 illustrates this method to establish an extremely important and useful logical equivalence, namely, that of  $\neg(p \vee q)$  with  $\neg p \wedge \neg q$ . This logical equivalence is one of the two **De Morgan laws**, shown in Table 2, named after the English mathematician Augustus De Morgan, of the mid-nineteenth century.

**EXAMPLE 2** Show that  $\neg(p \vee q)$  and  $\neg p \wedge \neg q$  are logically equivalent.

**Solution:** The truth tables for these compound propositions are displayed in Table 3. Because the truth values of the compound propositions  $\neg(p \vee q)$  and  $\neg p \wedge \neg q$  agree for all possible combinations of the truth values of  $p$  and  $q$ , it follows that  $\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$  is a tautology and that these compound propositions are logically equivalent.  $\blacktriangleleft$

**TABLE 3** Truth Tables for  $\neg(p \vee q)$  and  $\neg p \wedge \neg q$ .

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

**EXAMPLE 3** Show that  $p \rightarrow q$  and  $\neg p \vee q$  are logically equivalent.

**Solution:** We construct the truth table for these compound propositions in Table 4. Because the truth values of  $\neg p \vee q$  and  $p \rightarrow q$  agree, they are logically equivalent.  $\blacktriangleleft$

**TABLE 4** Truth Tables for  $\neg p \vee q$  and  $p \rightarrow q$ .

$p$	$q$	$\neg p$	$\neg p \vee q$	$p \rightarrow q$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

We will now establish a logical equivalence of two compound propositions involving three different propositional variables  $p$ ,  $q$ , and  $r$ . To use a truth table to establish such a logical equivalence, we need eight rows, one for each possible combination of truth values of these three variables. We symbolically represent these combinations by listing the truth values of  $p$ ,  $q$ , and  $r$ , respectively. These eight combinations of truth values are TTT, TT $F$ , TFT, T $F$ F, FTT, FTF, FFT, and FFF; we use this order when we display the rows of the truth table. Note that we need to double the number of rows in the truth tables we use to show that compound propositions are equivalent for each additional propositional variable, so that 16 rows are needed to establish the logical equivalence of two compound propositions involving four propositional variables, and so on. In general,  $2^n$  rows are required if a compound proposition involves  $n$  propositional variables.

**TABLE 5** A Demonstration That  $p \vee (q \wedge r)$  and  $(p \vee q) \wedge (p \vee r)$  Are Logically Equivalent.

$p$	$q$	$r$	$q \wedge r$	$p \vee (q \wedge r)$	$p \vee q$	$p \vee r$	$(p \vee q) \wedge (p \vee r)$
T	T	T	T	T	T	T	T
T	T	F	F	T	T	T	T
T	F	T	F	T	T	T	T
T	F	F	F	T	T	T	T
F	T	T	T	T	T	T	T
F	T	F	F	F	T	F	F
F	F	T	F	F	F	T	F
F	F	F	F	F	F	F	F

**EXAMPLE 4** Show that  $p \vee (q \wedge r)$  and  $(p \vee q) \wedge (p \vee r)$  are logically equivalent. This is the *distributive law* of disjunction over conjunction.

**Solution:** We construct the truth table for these compound propositions in Table 5. Because the truth values of  $p \vee (q \wedge r)$  and  $(p \vee q) \wedge (p \vee r)$  agree, these compound propositions are logically equivalent.  $\blacktriangleleft$

The identities in Table 6 are a special case of Boolean algebra identities found in Table 5 of Section 12.1. See Table 1 in Section 2.2 for analogous set identities.

Table 6 contains some important equivalences. In these equivalences, **T** denotes the compound proposition that is always true and **F** denotes the compound proposition that is always

**TABLE 6** Logical Equivalences.

Equivalence	Name
$p \wedge T = p$ $p \vee F = p$	Identity laws
$p \vee T = T$ $p \wedge F = F$	Domination laws
$p \vee p = p$ $p \wedge p = p$	Idempotent laws
$\neg(\neg p) = p$	Double negation law
$p \vee q = q \vee p$ $p \wedge q = q \wedge p$	Commutative laws
$(p \vee q) \vee r = p \vee (q \vee r)$ $(p \wedge q) \wedge r = p \wedge (q \wedge r)$	Associative laws
$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	Distributive laws
$\neg(p \wedge q) = \neg p \vee \neg q$ $\neg(p \vee q) = \neg p \wedge \neg q$	De Morgan's laws
$p \vee (p \wedge q) = p$ $p \wedge (p \vee q) = p$	Absorption laws
$p \vee \neg p = T$ $p \wedge \neg p = F$	Negation laws

**TABLE 7** Logical Equivalences Involving Conditional Statements.

$p \rightarrow q \equiv \neg p \vee q$
$p \rightarrow q \equiv \neg q \rightarrow \neg p$
$p \vee q \equiv \neg p \rightarrow q$
$p \wedge q \equiv \neg(p \rightarrow \neg q)$
$\neg(p \rightarrow q) \equiv p \wedge \neg q$
$(p \rightarrow q) \wedge (p \rightarrow r) = p \rightarrow (q \wedge r)$
$(p \rightarrow r) \wedge (q \rightarrow r) = (p \vee q) \rightarrow r$
$(p \rightarrow q) \vee (p \rightarrow r) = p \rightarrow (q \vee r)$
$(p \rightarrow r) \vee (q \rightarrow r) = (p \wedge q) \rightarrow r$

**TABLE 8** Logical Equivalences Involving Biconditional Statements.

$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$
$p \leftrightarrow q \equiv \neg p \leftrightarrow \neg q$
$p \leftrightarrow q \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$
$\neg(p \leftrightarrow q) = p \leftrightarrow \neg q$

false. We also display some useful equivalences for compound propositions involving conditional statements and biconditional statements in Tables 7 and 8, respectively. The reader is asked to verify the equivalences in Tables 6–8 in the exercises.

The associative law for disjunction shows that the expression  $p \vee q \vee r$  is well defined, in the sense that it does not matter whether we first take the disjunction of  $p$  with  $q$  and then the disjunction of  $p \vee q$  with  $r$ , or if we first take the disjunction of  $q$  and  $r$  and then take the disjunction of  $p$  with  $q \vee r$ . Similarly, the expression  $p \wedge q \wedge r$  is well defined. By extending this reasoning, it follows that  $p_1 \vee p_2 \vee \cdots \vee p_n$  and  $p_1 \wedge p_2 \wedge \cdots \wedge p_n$  are well defined whenever  $p_1, p_2, \dots, p_n$  are propositions.

Furthermore, note that De Morgan's laws extend to

$$\neg(p_1 \vee p_2 \vee \cdots \vee p_n) \equiv (\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_n)$$

and

$$\neg(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \equiv (\neg p_1 \vee \neg p_2 \vee \cdots \vee \neg p_n).$$

We will sometimes use the notation  $\bigvee_{j=1}^n p_j$  for  $p_1 \vee p_2 \vee \cdots \vee p_n$  and  $\bigwedge_{j=1}^n p_j$  for  $p_1 \wedge p_2 \wedge \cdots \wedge p_n$ . Using this notation, the extended version of De Morgan's laws can be written concisely as  $\neg(\bigvee_{j=1}^n p_j) \equiv \bigwedge_{j=1}^n \neg p_j$  and  $\neg(\bigwedge_{j=1}^n p_j) \equiv \bigvee_{j=1}^n \neg p_j$ . (Methods for proving these identities will be given in Section 5.1.)

## Using De Morgan's Laws

The two logical equivalences known as De Morgan's laws are particularly important. They tell us how to negate conjunctions and how to negate disjunctions. In particular, the equivalence  $\neg(p \vee q) \equiv \neg p \wedge \neg q$  tells us that the negation of a disjunction is formed by taking the conjunction of the negations of the component propositions. Similarly, the equivalence  $\neg(p \wedge q) \equiv \neg p \vee \neg q$  tells us that the negation of a conjunction is formed by taking the disjunction of the negations of the component propositions. Example 5 illustrates the use of De Morgan's laws.

When using De Morgan's laws, remember to change the logical connective after you negate.

**EXAMPLE 5** Use De Morgan's laws to express the negations of "Miguel has a cellphone and he has a laptop computer" and "Heather will go to the concert or Steve will go to the concert."



**Solution:** Let  $p$  be "Miguel has a cellphone" and  $q$  be "Miguel has a laptop computer." Then "Miguel has a cellphone and he has a laptop computer" can be represented by  $p \wedge q$ . By the first of De Morgan's laws,  $\neg(p \wedge q)$  is equivalent to  $\neg p \vee \neg q$ . Consequently, we can express the negation of our original statement as "Miguel does not have a cellphone or he does not have a laptop computer."

Let  $r$  be "Heather will go to the concert" and  $s$  be "Steve will go to the concert." Then "Heather will go to the concert or Steve will go to the concert" can be represented by  $r \vee s$ . By the second of De Morgan's laws,  $\neg(r \vee s)$  is equivalent to  $\neg r \wedge \neg s$ . Consequently, we can express the negation of our original statement as "Heather will not go to the concert and Steve will not go to the concert."

## Constructing New Logical Equivalences

The logical equivalences in Table 6, as well as any others that have been established (such as those shown in Tables 7 and 8), can be used to construct additional logical equivalences. The reason for this is that a proposition in a compound proposition can be replaced by a compound proposition that is logically equivalent to it without changing the truth value of the original compound proposition. This technique is illustrated in Examples 6–8, where we also use the fact that if  $p$  and  $q$  are logically equivalent and  $q$  and  $r$  are logically equivalent, then  $p$  and  $r$  are logically equivalent (see Exercise 56).

**EXAMPLE 6** Show that  $\neg(p \rightarrow q)$  and  $p \wedge \neg q$  are logically equivalent.



**Solution:** We could use a truth table to show that these compound propositions are equivalent (similar to what we did in Example 4). Indeed, it would not be hard to do so. However, we want to illustrate how to use logical identities that we already know to establish new logical identities, something that is of practical importance for establishing equivalences of compound propositions with a large number of variables. So, we will establish this equivalence by developing a series of



**AUGUSTUS DE MORGAN (1806–1871)** Augustus De Morgan was born in India, where his father was a colonel in the Indian army. De Morgan's family moved to England when he was 7 months old. He attended private schools, where in his early teens he developed a strong interest in mathematics. De Morgan studied at Trinity College, Cambridge, graduating in 1827. Although he considered medicine or law, he decided on mathematics for his career. He won a position at University College, London, in 1828, but resigned after the college dismissed a fellow professor without giving reasons. However, he resumed this position in 1836 when his successor died, remaining until 1866.

De Morgan was a noted teacher who stressed principles over techniques. His students included many famous mathematicians, including Augusta Ada, Countess of Lovelace, who was Charles Babbage's collaborator in his work on computing machines (see page 31 for biographical notes on Augusta Ada). (De Morgan cautioned the countess against studying too much mathematics, because it might interfere with her childbearing abilities!)

De Morgan was an extremely prolific writer, publishing more than 1000 articles in more than 15 periodicals. De Morgan also wrote textbooks on many subjects, including logic, probability, calculus, and algebra. In 1838 he presented what was perhaps the first clear explanation of an important proof technique known as *mathematical induction* (discussed in Section 5.1 of this text), a term he coined. In the 1840s De Morgan made fundamental contributions to the development of symbolic logic. He invented notations that helped him prove propositional equivalences, such as the laws that are named after him. In 1842 De Morgan presented what is considered to be the first precise definition of a limit and developed new tests for convergence of infinite series. De Morgan was also interested in the history of mathematics and wrote biographies of Newton and Halley.

In 1837 De Morgan married Sophia Frend, who wrote his biography in 1882. De Morgan's research, writing, and teaching left little time for his family or social life. Nevertheless, he was noted for his kindness, humor, and wide range of knowledge.

logical equivalences, using one of the equivalences in Table 6 at a time, starting with  $\neg(p \rightarrow q)$  and ending with  $p \wedge \neg q$ . We have the following equivalences.

$$\begin{aligned}\neg(p \rightarrow q) &\equiv \neg(\neg p \vee q) && \text{by Example 3} \\ &\equiv \neg(\neg p) \wedge \neg q && \text{by the second De Morgan law} \\ &\equiv p \wedge \neg q && \text{by the double negation law}\end{aligned}$$

**EXAMPLE 7** Show that  $\neg(p \vee (\neg p \wedge q))$  and  $\neg p \wedge \neg q$  are logically equivalent by developing a series of logical equivalences.

*Solution:* We will use one of the equivalences in Table 6 at a time, starting with  $\neg(p \vee (\neg p \wedge q))$  and ending with  $\neg p \wedge \neg q$ . (Note: we could also easily establish this equivalence using a truth table.) We have the following equivalences.

$$\begin{aligned}\neg(p \vee (\neg p \wedge q)) &\equiv \neg p \wedge \neg(\neg p \wedge q) && \text{by the second De Morgan law} \\ &\equiv \neg p \wedge [\neg(\neg p) \vee \neg q] && \text{by the first De Morgan law} \\ &\equiv \neg p \wedge (p \vee \neg q) && \text{by the double negation law} \\ &\equiv (\neg p \wedge p) \vee (\neg p \wedge \neg q) && \text{by the second distributive law} \\ &\equiv \mathbf{F} \vee (\neg p \wedge \neg q) && \text{because } \neg p \wedge p = \mathbf{F} \\ &\equiv (\neg p \wedge \neg q) \vee \mathbf{F} && \text{by the commutative law for disjunction} \\ &\equiv \neg p \wedge \neg q && \text{by the identity law for } \mathbf{F}\end{aligned}$$

Consequently  $\neg(p \vee (\neg p \wedge q))$  and  $\neg p \wedge \neg q$  are logically equivalent.

**EXAMPLE 8** Show that  $(p \wedge q) \rightarrow (p \vee q)$  is a tautology.

*Solution:* To show that this statement is a tautology, we will use logical equivalences to demonstrate that it is logically equivalent to  $\mathbf{T}$ . (Note: This could also be done using a truth table.)

$$\begin{aligned}(p \wedge q) \rightarrow (p \vee q) &\equiv \neg(p \wedge q) \vee (p \vee q) && \text{by Example 3} \\ &\equiv (\neg p \vee \neg q) \vee (p \vee q) && \text{by the first De Morgan law} \\ &\equiv (\neg p \vee p) \vee (\neg q \vee q) && \text{by the associative and commutative laws for disjunction} \\ &\equiv \mathbf{T} \vee \mathbf{T} && \text{by Example 1 and the commutative law for disjunction} \\ &\equiv \mathbf{T} && \text{by the domination law}\end{aligned}$$

## Propositional Satisfiability

A compound proposition is **satisfiable** if there is an assignment of truth values to its variables that makes it true. When no such assignments exists, that is, when the compound proposition is false for all assignments of truth values to its variables, the compound proposition is **unsatisfiable**.

Note that a compound proposition is unsatisfiable if and only if its negation is true for all assignments of truth values to the variables, that is, if and only if its negation is a tautology.

When we find a particular assignment of truth values that makes a compound proposition true, we have shown that it is satisfiable; such an assignment is called a **solution** of this particular

satisfiability problem. However, to show that a compound proposition is unsatisfiable, we need to show that *every* assignment of truth values to its variables makes it false. Although we can always use a truth table to determine whether a compound proposition is satisfiable, it is often more efficient not to, as Example 9 demonstrates.

**EXAMPLE 9** Determine whether each of the compound propositions  $(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p)$ ,  $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$ , and  $(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p) \wedge (p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$  is satisfiable.

**Solution:** Instead of using truth table to solve this problem, we will reason about truth values. Note that  $(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p)$  is true when the three variable  $p$ ,  $q$ , and  $r$  have the same truth value (see Exercise 40 of Section 1.1). Hence, it is satisfiable as there is at least one assignment of truth values for  $p$ ,  $q$ , and  $r$  that makes it true. Similarly, note that  $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$  is true when at least one of  $p$ ,  $q$ , and  $r$  is true and at least one is false (see Exercise 41 of Section 1.1). Hence,  $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$  is satisfiable, as there is at least one assignment of truth values for  $p$ ,  $q$ , and  $r$  that makes it true.

Finally, note that for  $(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p) \wedge (p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$  to be true,  $(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p)$  and  $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$  must both be true. For the first to be true, the three variables must have the same truth values, and for the second to be true, at least one of three variables must be true and at least one must be false. However, these conditions are contradictory. From these observations we conclude that no assignment of truth values to  $p$ ,  $q$ , and  $r$  makes  $(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p) \wedge (p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$  true. Hence, it is unsatisfiable.  $\blacktriangleleft$



**AUGUSTA ADA, COUNTESS OF LOVELACE (1815–1852)** Augusta Ada was the only child from the marriage of the famous poet Lord Byron and Lady Byron, Annabella Millbanke, who separated when Ada was 1 month old, because of Lord Byron's scandalous affair with his half sister. The Lord Byron had quite a reputation, being described by one of his lovers as "mad, bad, and dangerous to know." Lady Byron was noted for her intellect and had a passion for mathematics; she was called by Lord Byron "The Princess of Parallelograms." Augusta was raised by her mother, who encouraged her intellectual talents especially in music and mathematics, to counter what Lady Byron considered dangerous poetic tendencies. At this time, women were not allowed to attend universities and could not join learned societies. Nevertheless, Augusta pursued her mathematical studies independently and with mathematicians, including William Frend. She was also encouraged by another female mathematician, Mary Somerville, and in 1834 at a dinner party hosted by Mary Somerville, she learned about Charles Babbage's ideas for a calculating machine, called the Analytic Engine. In 1838 Augusta Ada married Lord King, later elevated to Earl of Lovelace. Together they had three children.

Augusta Ada continued her mathematical studies after her marriage. Charles Babbage had continued work on his Analytic Engine and lectured on this in Europe. In 1842 Babbage asked Augusta Ada to translate an article in French describing Babbage's invention. When Babbage saw her translation, he suggested she add her own notes, and the resulting work was three times the length of the original. The most complete accounts of the Analytic Engine are found in Augusta Ada's notes. In her notes, she compared the working of the Analytic Engine to that of the Jacquard loom, with Babbage's punch cards analogous to the cards used to create patterns on the loom. Furthermore, she recognized the promise of the machine as a general purpose computer much better than Babbage did. She stated that the "engine is the material expression of any indefinite function of any degree of generality and complexity." Her notes on the Analytic Engine anticipate many future developments, including computer-generated music. Augusta Ada published her writings under her initials A.A.L. concealing her identity as a woman as did many women at a time when women were not considered to be the intellectual equals of men. After 1845 she and Babbage worked toward the development of a system to predict horse races. Unfortunately, their system did not work well, leaving Augusta Ada heavily in debt at the time of her death at an unfortunately young age from uterine cancer.

In 1953 Augusta Ada's notes on the Analytic Engine were republished more than 100 years after they were written, and after they had been long forgotten. In his work in the 1950s on the capacity of computers to think (and his famous Turing Test), Alan Turing responded to Augusta Ada's statement that "The Analytic Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform." This "dialogue" between Turing and Augusta Ada is still the subject of controversy. Because of her fundamental contributions to computing, the programming language Ada is named in honor of the Countess of Lovelace.

	2	9				4		
			5			1		
4								
			4	2				
6						7		
5								
7		3				5		
1			9					
						6		

**FIGURE 1** A  $9 \times 9$  Sudoku puzzle.

## Applications of Satisfiability

Many problems, in diverse areas such as robotics, software testing, computer-aided design, machine vision, integrated circuit design, computer networking, and genetics, can be modeled in terms of propositional satisfiability. Although most of these applications are beyond the scope of this book, we will study one application here. In particular, we will show how to use propositional satisfiability to model Sudoku puzzles.



**SUDOKU** A **Sudoku puzzle** is represented by a  $9 \times 9$  grid made up of nine  $3 \times 3$  subgrids, known as **blocks**, as shown in Figure 1. For each puzzle, some of the 81 cells, called **givens**, are assigned one of the numbers 1, 2, . . . , 9, and the other cells are blank. The puzzle is solved by assigning a number to each blank cell so that every row, every column, and every one of the nine  $3 \times 3$  blocks contains each of the nine possible numbers. Note that instead of using a  $9 \times 9$  grid, Sudoku puzzles can be based on  $n^2 \times n^2$  grids, for any positive integer  $n$ , with the  $n^2 \times n^2$  grid made up of  $n^2 n \times n$  subgrids.

The popularity of Sudoku dates back to the 1980s when it was introduced in Japan. It took 20 years for Sudoku to spread to rest of the world, but by 2005, Sudoku puzzles were a worldwide craze. The name Sudoku is short for the Japanese *suuji wa dokushin ni kagiru*, which means “the digits must remain single.” The modern game of Sudoku was apparently designed in the late 1970s by an American puzzle designer. The basic ideas of Sudoku date back even further; puzzles printed in French newspapers in the 1890s were quite similar, but not identical, to modern Sudoku.

Sudoku puzzles designed for entertainment have two additional important properties. First, they have exactly one solution. Second, they can be solved using reasoning alone, that is, without resorting to searching all possible assignments of numbers to the cells. As a Sudoku puzzle is solved, entries in blank cells are successively determined by already known values. For instance, in the grid in Figure 1, the number 4 must appear in exactly one cell in the second row. How can we determine which of the seven blank cells it must appear? First, we observe that 4 cannot appear in one of the first three cells or in one of the last three cells of this row, because it already appears in another cell in the block each of these cells is in. We can also see that 4 cannot appear in the fifth cell in this row, as it already appears in the fifth column in the fourth row. This means that 4 must appear in the sixth cell of the second row.

Many strategies based on logic and mathematics have been devised for solving Sudoku puzzles (see [Da10], for example). Here, we discuss one of the ways that have been developed for solving Sudoku puzzles with the aid of a computer, which depends on modeling the puzzle as a propositional satisfiability problem. Using the model we describe, particular Sudoku puzzles can be solved using software developed to solve satisfiability problems. Currently, Sudoku puzzles can be solved in less than 10 milliseconds this way. It should be noted that there are many other approaches for solving Sudoku puzzles via computers using other techniques.

To encode a Sudoku puzzle, let  $p(i, j, n)$  denote the proposition that is true when the number  $n$  is in the cell in the  $i$ th row and  $j$ th column. There are  $9 \times 9 \times 9 = 729$  such propositions, as  $i$ ,  $j$ , and  $n$  all range from 1 to 9. For example, for the puzzle in Figure 1, the number 6 is given as the value in the fifth row and first column. Hence, we see that  $p(5, 1, 6)$  is true, but  $p(5, j, 6)$  is false for  $j = 2, 3, \dots, 9$ .

Given a particular Sudoku puzzle, we begin by encoding each of the given values. Then, we construct compound propositions that assert that every row contains every number, every column contains every number, every  $3 \times 3$  block contains every number, and each cell contains no more than one number. It follows, as the reader should verify, that the Sudoku puzzle is solved by finding an assignment of truth values to the 729 propositions  $p(i, j, n)$  with  $i$ ,  $j$ , and  $n$  each ranging from 1 to 9 that makes the conjunction of all these compound propositions true. After listing these assertions, we will explain how to construct the assertion that every row contains every integer from 1 to 9. We will leave the construction of the other assertions that every column contains every number and each of the nine  $3 \times 3$  blocks contains every number to the exercises.

- For each cell with a given value, we assert  $p(i, j, n)$  when the cell in row  $i$  and column  $j$  has the given value  $n$ .
- We assert that every row contains every number:

$$\bigwedge_{i=1}^9 \bigwedge_{n=1}^9 \bigvee_{j=1}^9 p(i, j, n)$$

- We assert that every column contains every number:

$$\bigwedge_{j=1}^9 \bigwedge_{n=1}^9 \bigvee_{i=1}^9 p(i, j, n)$$



It is tricky setting up the two inner indices so that all nine cells in each square block are examined.

- We assert that each of the nine  $3 \times 3$  blocks contains every number:

$$\bigwedge_{r=0}^2 \bigwedge_{s=0}^2 \bigwedge_{n=1}^9 \bigvee_{i=1}^3 \bigvee_{j=1}^3 p(3r+i, 3s+j, n)$$

- To assert that no cell contains more than one number, we take the conjunction over all values of  $n, n', i$ , and  $j$  where each variable ranges from 1 to 9 and  $n \neq n'$  of  $p(i, j, n) \rightarrow \neg p(i, j, n')$ .

We now explain how to construct the assertion that every row contains every number. First, to assert that row  $i$  contains the number  $n$ , we form  $\bigvee_{j=1}^9 p(i, j, n)$ . To assert that row  $i$  contains all  $n$  numbers, we form the conjunction of these disjunctions over all nine possible values of  $n$ , giving us  $\bigwedge_{n=1}^9 \bigvee_{j=1}^9 p(i, j, n)$ . Finally, to assert that every row contains every number, we take the conjunction of  $\bigwedge_{n=1}^9 \bigvee_{j=1}^9 p(i, j, n)$  over all nine rows. This gives us  $\bigwedge_{i=1}^9 \bigwedge_{n=1}^9 \bigvee_{j=1}^9 p(i, j, n)$ . (Exercises 65 and 66 ask for explanations of the assertions that every column contains every number and that each of the nine  $3 \times 3$  blocks contains every number.)

Given a particular Sudoku puzzle, to solve this puzzle we can find a solution to the satisfiability problems that asks for a set of truth values for the 729 variables  $p(i, j, n)$  that makes the conjunction of all the listed assertions true.

## Solving Satisfiability Problems

A truth table can be used to determine whether a compound proposition is satisfiable, or equivalently, whether its negation is a tautology (see Exercise 60). This can be done by hand for a compound proposition with a small number of variables, but when the number of variables grows, this becomes impractical. For instance, there are  $2^{20} = 1,048,576$  rows in the truth table for a compound proposition with 20 variables. Clearly, you need a computer to help you determine, in this way, whether a compound proposition in 20 variables is satisfiable.

When many applications are modeled, questions concerning the satisfiability of compound propositions with hundreds, thousands, or millions of variables arise. Note, for example, that when there are 1000 variables, checking every one of the  $2^{1000}$  (a number with more than 300 decimal digits) possible combinations of truth values of the variables in a compound proposition cannot be done by a computer in even trillions of years. No procedure is known that a computer can follow to determine in a reasonable amount of time whether an arbitrary compound proposition in such a large number of variables is satisfiable. However, progress has been made developing methods for solving the satisfiability problem for the particular types of compound propositions that arise in practical applications, such as for the solution of Sudoku puzzles. Many computer programs have been developed for solving satisfiability problems which have practical use. In our discussion of the subject of algorithms in Chapter 3, we will discuss this question further. In particular, we will explain the important role the propositional satisfiability problem plays in the study of the complexity of algorithms.



### Exercises

1. Use truth tables to verify these equivalences.
 

a) $p \wedge T \equiv p$	b) $p \vee F \equiv p$
c) $p \wedge F \equiv F$	d) $p \vee T \equiv T$
e) $p \vee p \equiv p$	f) $p \wedge p \equiv p$
2. Show that  $\neg(\neg p)$  and  $p$  are logically equivalent.
3. Use truth tables to verify the commutative laws
 

a) $p \vee q \equiv q \vee p$ .	b) $p \wedge q \equiv q \wedge p$ .
---------------------------------	-------------------------------------
4. Use truth tables to verify the associative laws
 

a) $(p \vee q) \vee r \equiv p \vee (q \vee r)$ .	b) $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ .
---	---
5. Use a truth table to verify the distributive law  

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r).$$
6. Use a truth table to verify the first De Morgan law  

$$\neg(p \wedge q) \equiv \neg p \vee \neg q.$$
7. Use De Morgan's laws to find the negation of each of the following statements.
 

a) Jan is rich and happy.
b) Carlos will bicycle or run tomorrow.



**HENRY MAURICE SHEFFER (1883–1964)** Henry Maurice Sheffer, born to Jewish parents in the western Ukraine, emigrated to the United States in 1892 with his parents and six siblings. He studied at the Boston Latin School before entering Harvard, where he completed his undergraduate degree in 1905, his master's in 1907, and his Ph.D. in philosophy in 1908. After holding a postdoctoral position at Harvard, Henry traveled to Europe on a fellowship. Upon returning to the United States, he became an academic nomad, spending one year each at the University of Washington, Cornell, the University of Minnesota, the University of Missouri, and City College in New York. In 1916 he returned to Harvard as a faculty member in the philosophy department. He remained at Harvard until his retirement in 1952.

Sheffer introduced what is now known as the Sheffer stroke in 1913; it became well known only after its use in the 1925 edition of Whitehead and Russell's *Principia Mathematica*. In this same edition Russell wrote that Sheffer had invented a powerful method that could be used to simplify the *Principia*. Because of this comment, Sheffer was something of a mystery man to logicians, especially because Sheffer, who published little in his career, never published the details of this method, only describing it in mimeographed notes and in a brief published abstract.

Sheffer was a dedicated teacher of mathematical logic. He liked his classes to be small and did not like auditors. When strangers appeared in his classroom, Sheffer would order them to leave, even his colleagues or distinguished guests visiting Harvard. Sheffer was barely five feet tall; he was noted for his wit and vigor, as well as for his nervousness and irritability. Although widely liked, he was quite lonely. He is noted for a quip he spoke at his retirement: "Old professors never die, they just become emeriti." Sheffer is also credited with coining the term "Boolean algebra" (the subject of Chapter 12 of this text). Sheffer was briefly married and lived most of his later life in small rooms at a hotel packed with his logic books and vast files of slips of paper he used to jot down his ideas. Unfortunately, Sheffer suffered from severe depression during the last two decades of his life.

- c) Mei walks or takes the bus to class.  
 d) Ibrahim is smart and hard working.
8. Use De Morgan's laws to find the negation of each of the following statements.
- Kwame will take a job in industry or go to graduate school.
  - Yoshiko knows Java and calculus.
  - James is young and strong.
  - Rita will move to Oregon or Washington.
9. Show that each of these conditional statements is a tautology by using truth tables.
- $(p \wedge q) \rightarrow p$
  - $p \rightarrow (p \vee q)$
  - $\neg p \rightarrow (p \rightarrow q)$
  - $(p \wedge q) \rightarrow (p \rightarrow q)$
  - $\neg(p \rightarrow q) \rightarrow p$
  - $\neg(p \rightarrow q) \rightarrow \neg q$
10. Show that each of these conditional statements is a tautology by using truth tables.
- $[\neg p \wedge (p \vee q)] \rightarrow q$
  - $[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$
  - $[p \wedge (p \rightarrow q)] \rightarrow q$
  - $[(p \vee q) \wedge (p \rightarrow r) \wedge (q \rightarrow r)] \rightarrow r$
11. Show that each conditional statement in Exercise 9 is a tautology without using truth tables.
12. Show that each conditional statement in Exercise 10 is a tautology without using truth tables.
13. Use truth tables to verify the absorption laws.
- $p \vee (p \wedge q) \equiv p$
  - $p \wedge (p \vee q) \equiv p$
14. Determine whether  $(\neg p \wedge (p \rightarrow q)) \rightarrow \neg q$  is a tautology.
15. Determine whether  $(\neg q \wedge (p \rightarrow q)) \rightarrow \neg p$  is a tautology.
- Each of Exercises 16–28 asks you to show that two compound propositions are logically equivalent. To do this, either show that both sides are true, or that both sides are false, for exactly the same combinations of truth values of the propositional variables in these expressions (whichever is easier).
- Show that  $p \leftrightarrow q$  and  $(p \wedge q) \vee (\neg p \wedge \neg q)$  are logically equivalent.
  - Show that  $\neg(p \leftrightarrow q)$  and  $p \leftrightarrow \neg q$  are logically equivalent.
  - Show that  $p \rightarrow q$  and  $\neg q \rightarrow \neg p$  are logically equivalent.
  - Show that  $\neg p \leftrightarrow q$  and  $p \leftrightarrow \neg q$  are logically equivalent.
  - Show that  $\neg(p \oplus q)$  and  $p \leftrightarrow q$  are logically equivalent.
  - Show that  $\neg(p \leftrightarrow q)$  and  $\neg p \leftrightarrow q$  are logically equivalent.
  - Show that  $(p \rightarrow q) \wedge (p \rightarrow r)$  and  $p \rightarrow (q \wedge r)$  are logically equivalent.
  - Show that  $(p \rightarrow r) \wedge (q \rightarrow r)$  and  $(p \vee q) \rightarrow r$  are logically equivalent.
  - Show that  $(p \rightarrow q) \vee (p \rightarrow r)$  and  $p \rightarrow (q \vee r)$  are logically equivalent.
  - Show that  $(p \rightarrow r) \vee (q \rightarrow r)$  and  $(p \wedge q) \rightarrow r$  are logically equivalent.
  - Show that  $\neg p \rightarrow (q \rightarrow r)$  and  $q \rightarrow (p \vee r)$  are logically equivalent.
  - Show that  $p \leftrightarrow q$  and  $(p \rightarrow q) \wedge (q \rightarrow p)$  are logically equivalent.
  - Show that  $p \leftrightarrow q$  and  $\neg p \leftrightarrow \neg q$  are logically equivalent.

29. Show that  $(p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r)$  is a tautology.

30. Show that  $(p \vee q) \wedge (\neg p \vee r) \rightarrow (q \vee r)$  is a tautology.

31. Show that  $(p \rightarrow q) \rightarrow r$  and  $p \rightarrow (q \rightarrow r)$  are not logically equivalent.

32. Show that  $(p \wedge q) \rightarrow r$  and  $(p \rightarrow r) \wedge (q \rightarrow r)$  are not logically equivalent.

33. Show that  $(p \rightarrow q) \rightarrow (r \rightarrow s)$  and  $(p \rightarrow r) \rightarrow (q \rightarrow s)$  are not logically equivalent.

The **dual** of a compound proposition that contains only the logical operators  $\vee$ ,  $\wedge$ , and  $\neg$  is the compound proposition obtained by replacing each  $\vee$  by  $\wedge$ , each  $\wedge$  by  $\vee$ , each **T** by **F**, and each **F** by **T**. The dual of  $s$  is denoted by  $s^*$ .

34. Find the dual of each of these compound propositions.

- $p \vee \neg q$
- $p \wedge (q \vee (r \wedge T))$
- $(p \wedge \neg q) \vee (q \wedge F)$

35. Find the dual of each of these compound propositions.

- $p \wedge \neg q \wedge \neg r$
- $(p \wedge q \wedge r) \vee s$
- $(p \vee F) \wedge (q \vee T)$

36. When does  $s^* = s$ , where  $s$  is a compound proposition?

37. Show that  $(s^*)^* = s$  when  $s$  is a compound proposition.

38. Show that the logical equivalences in Table 6, except for the double negation law, come in pairs, where each pair contains compound propositions that are duals of each other.

- \*\*39.** Why are the duals of two equivalent compound propositions also equivalent, where these compound propositions contain only the operators  $\wedge$ ,  $\vee$ , and  $\neg$ ?

40. Find a compound proposition involving the propositional variables  $p$ ,  $q$ , and  $r$  that is true when  $p$  and  $q$  are true and  $r$  is false, but is false otherwise. [Hint: Use a conjunction of each propositional variable or its negation.]

41. Find a compound proposition involving the propositional variables  $p$ ,  $q$ , and  $r$  that is true when exactly two of  $p$ ,  $q$ , and  $r$  are true and is false otherwise. [Hint: Form a disjunction of conjunctions. Include a conjunction for each combination of values for which the compound proposition is true. Each conjunction should include each of the three propositional variables or its negations.]

42. Suppose that a truth table in  $n$  propositional variables is specified. Show that a compound proposition with this truth table can be formed by taking the disjunction of conjunctions of the variables or their negations, with one conjunction included for each combination of values for which the compound proposition is true. The resulting compound proposition is said to be in **disjunctive normal form**.

A collection of logical operators is called **functionally complete** if every compound proposition is logically equivalent to a compound proposition involving only these logical operators.

43. Show that  $\neg$ ,  $\wedge$ , and  $\vee$  form a functionally complete collection of logical operators. [Hint: Use the fact that every compound proposition is logically equivalent to one in disjunctive normal form, as shown in Exercise 42.]

- \*44. Show that  $\neg$  and  $\wedge$  form a functionally complete collection of logical operators. [Hint: First use a De Morgan law to show that  $p \vee q$  is logically equivalent to  $\neg(\neg p \wedge \neg q)$ .]
- \*45. Show that  $\neg$  and  $\vee$  form a functionally complete collection of logical operators.

The following exercises involve the logical operators *NAND* and *NOR*. The proposition  $p \text{ NAND } q$  is true when either  $p$  or  $q$ , or both, are false; and it is false when both  $p$  and  $q$  are true. The proposition  $p \text{ NOR } q$  is true when both  $p$  and  $q$  are false, and it is false otherwise. The propositions  $p \text{ NAND } q$  and  $p \text{ NOR } q$  are denoted by  $p \mid q$  and  $p \downarrow q$ , respectively. (The operators  $\mid$  and  $\downarrow$  are called the **Sheffer stroke** and the **Peirce arrow** after H. M. Sheffer and C. S. Peirce, respectively.)

46. Construct a truth table for the logical operator *NAND*.
47. Show that  $p \mid q$  is logically equivalent to  $\neg(p \wedge q)$ .
48. Construct a truth table for the logical operator *NOR*.
49. Show that  $p \downarrow q$  is logically equivalent to  $\neg(p \vee q)$ .
50. In this exercise we will show that  $\{\downarrow\}$  is a functionally complete collection of logical operators.
- Show that  $p \downarrow p$  is logically equivalent to  $\neg p$ .
  - Show that  $(p \downarrow q) \downarrow (p \downarrow q)$  is logically equivalent to  $p \vee q$ .
  - Conclude from parts (a) and (b), and Exercise 49, that  $\{\downarrow\}$  is a functionally complete collection of logical operators.
- \*51. Find a compound proposition logically equivalent to  $p \rightarrow q$  using only the logical operator  $\downarrow$ .
52. Show that  $\{\mid\}$  is a functionally complete collection of logical operators.
53. Show that  $p \mid q$  and  $q \mid p$  are equivalent.
54. Show that  $p \mid (q \mid r)$  and  $(p \mid q) \mid r$  are not equivalent, so that the logical operator  $\mid$  is not associative.
- \*55. How many different truth tables of compound propositions are there that involve the propositional variables  $p$  and  $q$ ?
56. Show that if  $p$ ,  $q$ , and  $r$  are compound propositions such that  $p$  and  $q$  are logically equivalent and  $q$  and  $r$  are logically equivalent, then  $p$  and  $r$  are logically equivalent.
57. The following sentence is taken from the specification of a telephone system: "If the directory database is opened, then the monitor is put in a closed state, if the system is not in its initial state." This specification is hard to under-

stand because it involves two conditional statements. Find an equivalent, easier-to-understand specification that involves disjunctions and negations but not conditional statements.

58. How many of the disjunctions  $p \vee \neg q$ ,  $\neg p \vee q$ ,  $q \vee r$ ,  $q \vee \neg r$ , and  $\neg q \vee \neg r$  can be made simultaneously true by an assignment of truth values to  $p$ ,  $q$ , and  $r$ ?
59. How many of the disjunctions  $p \vee \neg q \vee s$ ,  $\neg p \vee \neg r \vee s$ ,  $\neg p \vee r \vee \neg s$ ,  $q \vee r \vee \neg s$ ,  $q \vee \neg r \vee \neg s$ ,  $\neg p \vee \neg q \vee \neg s$ ,  $p \vee r \vee s$ , and  $p \vee r \vee \neg s$  can be made simultaneously true by an assignment of truth values to  $p$ ,  $q$ ,  $r$ , and  $s$ ?
60. Show that the negation of an unsatisfiable compound proposition is a tautology and the negation of a compound proposition that is a tautology is unsatisfiable.
61. Determine whether each of these compound propositions is satisfiable.
- $(p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$
  - $(p \rightarrow q) \wedge (p \rightarrow \neg q) \wedge (\neg p \rightarrow q) \wedge (\neg p \rightarrow \neg q)$
  - $(p \leftrightarrow q) \wedge (\neg p \leftrightarrow q)$
62. Determine whether each of these compound propositions is satisfiable.
- $(p \vee q \vee \neg r) \wedge (p \vee \neg q \vee \neg s) \wedge (p \vee \neg r \vee \neg s) \wedge (\neg p \vee \neg q \vee \neg s) \wedge (p \vee q \vee \neg s)$
  - $(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q \vee \neg s) \wedge (p \vee \neg q \vee \neg s) \wedge (\neg p \vee \neg r \vee \neg s) \wedge (p \vee q \vee \neg r) \wedge (p \vee \neg r \vee \neg s)$
  - $(p \vee q \vee r) \wedge (p \vee \neg q \vee \neg s) \wedge (q \vee \neg r \vee s) \wedge (\neg p \vee r \vee s) \wedge (\neg p \vee q \vee \neg s) \wedge (p \vee \neg q \vee \neg r) \wedge (\neg p \vee \neg q \vee s) \wedge (\neg p \vee \neg r \vee \neg s)$
63. Show how the solution of a given  $4 \times 4$  Sudoku puzzle can be found by solving a satisfiability problem.
64. Construct a compound proposition that asserts that every cell of a  $9 \times 9$  Sudoku puzzle contains at least one number.
65. Explain the steps in the construction of the compound proposition given in the text that asserts that every column of a  $9 \times 9$  Sudoku puzzle contains every number.
- \*66. Explain the steps in the construction of the compound proposition given in the text that asserts that each of the nine  $3 \times 3$  blocks of a  $9 \times 9$  Sudoku puzzle contains every number.

## 1.4 Predicates and Quantifiers

### Introduction

Propositional logic, studied in Sections 1.1–1.3, cannot adequately express the meaning of all statements in mathematics and in natural language. For example, suppose that we know that

"Every computer connected to the university network is functioning properly."

## 1.6 Rules of Inference

---

### Introduction

Later in this chapter we will study proofs. Proofs in mathematics are valid arguments that establish the truth of mathematical statements. By an **argument**, we mean a sequence of statements that end with a conclusion. By **valid**, we mean that the conclusion, or final statement of the argument, must follow from the truth of the preceding statements, or **premises**, of the argument. That is, an argument is valid if and only if it is impossible for all the premises to be true and the conclusion to be false. To deduce new statements from statements we already have, we use rules of inference which are templates for constructing valid arguments. Rules of inference are our basic tools for establishing the truth of statements.

Before we study mathematical proofs, we will look at arguments that involve only compound propositions. We will define what it means for an argument involving compound propositions to be valid. Then we will introduce a collection of rules of inference in propositional logic. These rules of inference are among the most important ingredients in producing valid arguments. After we illustrate how rules of inference are used to produce valid arguments, we will describe some common forms of incorrect reasoning, called **fallacies**, which lead to invalid arguments.

After studying rules of inference in propositional logic, we will introduce rules of inference for quantified statements. We will describe how these rules of inference can be used to produce valid arguments. These rules of inference for statements involving existential and universal quantifiers play an important role in proofs in computer science and mathematics, although they are often used without being explicitly mentioned.

Finally, we will show how rules of inference for propositions and for quantified statements can be combined. These combinations of rule of inference are often used together in complicated arguments.

### Valid Arguments in Propositional Logic

Consider the following argument involving propositions (which, by definition, is a sequence of propositions):

“If you have a current password, then you can log onto the network.”

“You have a current password.”

Therefore,

“You can log onto the network.”

We would like to determine whether this is a valid argument. That is, we would like to determine whether the conclusion “You can log onto the network” must be true when the premises “If you have a current password, then you can log onto the network” and “You have a current password” are both true.

Before we discuss the validity of this particular argument, we will look at its form. Use  $p$  to represent “You have a current password” and  $q$  to represent “You can log onto the network.” Then, the argument has the form

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

where  $\therefore$  is the symbol that denotes “therefore.”

We know that when  $p$  and  $q$  are propositional variables, the statement  $((p \rightarrow q) \wedge p) \rightarrow q$  is a tautology (see Exercise 10(c) in Section 1.3). In particular, when both  $p \rightarrow q$  and  $p$  are true, we know that  $q$  must also be true. We say this form of argument is **valid** because whenever all its premises (all statements in the argument other than the final one, the conclusion) are true, the conclusion must also be true. Now suppose that both “If you have a current password, then you can log onto the network” and “You have a current password” are true statements. When we replace  $p$  by “You have a current password” and  $q$  by “You can log onto the network,” it necessarily follows that the conclusion “You can log onto the network” is true. This argument is **valid** because its form is valid. Note that whenever we replace  $p$  and  $q$  by propositions where  $p \rightarrow q$  and  $p$  are both true, then  $q$  must also be true.

What happens when we replace  $p$  and  $q$  in this argument form by propositions where not both  $p$  and  $p \rightarrow q$  are true? For example, suppose that  $p$  represents “You have access to the network” and  $q$  represents “You can change your grade” and that  $p$  is true, but  $p \rightarrow q$  is false. The argument we obtain by substituting these values of  $p$  and  $q$  into the argument form is

$$\begin{array}{l} \text{“If you have access to the network, then you can change your grade.”} \\ \text{“You have access to the network.”} \\ \hline \therefore \text{“You can change your grade.”} \end{array}$$

The argument we obtained is a valid argument, but because one of the premises, namely the first premise, is false, we cannot conclude that the conclusion is true. (Most likely, this conclusion is false.)

In our discussion, to analyze an argument, we replaced propositions by propositional variables. This changed an argument to an **argument form**. We saw that the validity of an argument follows from the validity of the form of the argument. We summarize the terminology used to discuss the validity of arguments with our definition of the key notions.

### DEFINITION 1

An *argument* in propositional logic is a sequence of propositions. All but the final proposition in the argument are called *premises* and the final proposition is called the *conclusion*. An argument is *valid* if the truth of all its premises implies that the conclusion is true.

An *argument form* in propositional logic is a sequence of compound propositions involving propositional variables. An argument form is *valid* no matter which particular propositions are substituted for the propositional variables in its premises, the conclusion is true if the premises are all true.

From the definition of a valid argument form we see that the argument form with premises  $p_1, p_2, \dots, p_n$  and conclusion  $q$  is valid, when  $(p_1 \wedge p_2 \wedge \dots \wedge p_n) \rightarrow q$  is a tautology.

The key to showing that an argument in propositional logic is valid is to show that its argument form is valid. Consequently, we would like techniques to show that argument forms are valid. We will now develop methods for accomplishing this task.

## Rules of Inference for Propositional Logic

We can always use a truth table to show that an argument form is valid. We do this by showing that whenever the premises are true, the conclusion must also be true. However, this can be a tedious approach. For example, when an argument form involves 10 different propositional variables, to use a truth table to show this argument form is valid requires  $2^{10} = 1024$  different rows. Fortunately, we do not have to resort to truth tables. Instead, we can first establish the validity of some relatively simple argument forms, called **rules of inference**. These rules of inference can be used as building blocks to construct more complicated valid argument forms. We will now introduce the most important rules of inference in propositional logic.

The tautology  $(p \wedge (p \rightarrow q)) \rightarrow q$  is the basis of the rule of inference called **modus ponens**, or the **law of detachment**. (Modus ponens is Latin for *mode that affirms*.) This tautology leads to the following valid argument form, which we have already seen in our initial discussion about arguments (where, as before, the symbol  $\therefore$  denotes “therefore”):

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

Using this notation, the hypotheses are written in a column, followed by a horizontal bar, followed by a line that begins with the therefore symbol and ends with the conclusion. In particular, modus ponens tells us that if a conditional statement and the hypothesis of this conditional statement are both true, then the conclusion must also be true. Example 1 illustrates the use of modus ponens.

**EXAMPLE 1** Suppose that the conditional statement “If it snows today, then we will go skiing” and its hypothesis, “It is snowing today,” are true. Then, by modus ponens, it follows that the conclusion of the conditional statement, “We will go skiing,” is true. 

As we mentioned earlier, a valid argument can lead to an incorrect conclusion if one or more of its premises is false. We illustrate this again in Example 2.

**EXAMPLE 2** Determine whether the argument given here is valid and determine whether its conclusion must be true because of the validity of the argument.

“If  $\sqrt{2} > \frac{3}{2}$ , then  $(\sqrt{2})^2 > (\frac{3}{2})^2$ . We know that  $\sqrt{2} > \frac{3}{2}$ . Consequently,  $(\sqrt{2})^2 = 2 > (\frac{3}{2})^2 = \frac{9}{4}$ .”

**Solution:** Let  $p$  be the proposition “ $\sqrt{2} > \frac{3}{2}$ ” and  $q$  the proposition “ $2 > (\frac{3}{2})^2$ .” The premises of the argument are  $p \rightarrow q$  and  $p$ , and  $q$  is its conclusion. This argument is valid because it is constructed by using modus ponens, a valid argument form. However, one of its premises,  $\sqrt{2} > \frac{3}{2}$ , is false. Consequently, we cannot conclude that the conclusion is true. Furthermore, note that the conclusion of this argument is false, because  $2 < \frac{9}{4}$ . 

There are many useful rules of inference for propositional logic. Perhaps the most widely used of these are listed in Table 1. Exercises 9, 10, 15, and 30 in Section 1.3 ask for the verifications that these rules of inference are valid argument forms. We now give examples of arguments that use these rules of inference. In each argument, we first use propositional variables to express the propositions in the argument. We then show that the resulting argument form is a rule of inference from Table 1.

**TABLE 1 Rules of Inference.**

<i>Rule of Inference</i>	<i>Tautology</i>	<i>Name</i>
$\begin{array}{l} p \\ p \rightarrow q \\ \hline \therefore q \end{array}$	$(p \wedge (p \rightarrow q)) \rightarrow q$	Modus ponens
$\begin{array}{l} \neg q \\ p \rightarrow q \\ \hline \therefore \neg p \end{array}$	$(\neg q \wedge (p \rightarrow q)) \rightarrow \neg p$	Modus tollens
$\begin{array}{l} p \rightarrow q \\ q \rightarrow r \\ \hline \therefore p \rightarrow r \end{array}$	$((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$	Hypothetical syllogism
$\begin{array}{l} p \vee q \\ \neg p \\ \hline \therefore q \end{array}$	$((p \vee q) \wedge \neg p) \rightarrow q$	Disjunctive syllogism
$\begin{array}{l} p \\ \hline \therefore p \vee q \end{array}$	$p \rightarrow (p \vee q)$	Addition
$\begin{array}{l} p \wedge q \\ \hline \therefore p \end{array}$	$(p \wedge q) \rightarrow p$	Simplification
$\begin{array}{l} p \\ q \\ \hline \therefore p \wedge q \end{array}$	$((p) \wedge (q)) \rightarrow (p \wedge q)$	Conjunction
$\begin{array}{l} p \vee q \\ \neg p \vee r \\ \hline \therefore q \vee r \end{array}$	$((p \vee q) \wedge (\neg p \vee r)) \rightarrow (q \vee r)$	Resolution

**EXAMPLE 3** State which rule of inference is the basis of the following argument: “It is below freezing now. Therefore, it is either below freezing or raining now.”

*Solution:* Let  $p$  be the proposition “It is below freezing now” and  $q$  the proposition “It is raining now.” Then this argument is of the form

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

This is an argument that uses the addition rule. ◀

**EXAMPLE 4** State which rule of inference is the basis of the following argument: “It is below freezing and raining now. Therefore, it is below freezing now.”

*Solution:* Let  $p$  be the proposition “It is below freezing now,” and let  $q$  be the proposition “It is raining now.” This argument is of the form

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

This argument uses the simplification rule. ◀

**EXAMPLE 5** State which rule of inference is used in the argument:

If it rains today, then we will not have a barbecue today. If we do not have a barbecue today, then we will have a barbecue tomorrow. Therefore, if it rains today, then we will have a barbecue tomorrow.

**Solution:** Let  $p$  be the proposition “It is raining today,” let  $q$  be the proposition “We will not have a barbecue today,” and let  $r$  be the proposition “We will have a barbecue tomorrow.” Then this argument is of the form

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

Hence, this argument is a hypothetical syllogism. 

### Using Rules of Inference to Build Arguments

When there are many premises, several rules of inference are often needed to show that an argument is valid. This is illustrated by Examples 6 and 7, where the steps of arguments are displayed on separate lines, with the reason for each step explicitly stated. These examples also show how arguments in English can be analyzed using rules of inference.

**EXAMPLE 6** Show that the premises “It is not sunny this afternoon and it is colder than yesterday,” “We will go swimming only if it is sunny,” “If we do not go swimming, then we will take a canoe trip,” and “If we take a canoe trip, then we will be home by sunset” lead to the conclusion “We will be home by sunset.”



**Solution:** Let  $p$  be the proposition “It is sunny this afternoon,”  $q$  the proposition “It is colder than yesterday,”  $r$  the proposition “We will go swimming,”  $s$  the proposition “We will take a canoe trip,” and  $t$  the proposition “We will be home by sunset.” Then the premises become  $\neg p \wedge q$ ,  $r \rightarrow p$ ,  $\neg r \rightarrow s$ , and  $s \rightarrow t$ . The conclusion is simply  $t$ . We need to give a valid argument with premises  $\neg p \wedge q$ ,  $r \rightarrow p$ ,  $\neg r \rightarrow s$ , and  $s \rightarrow t$  and conclusion  $t$ .

We construct an argument to show that our premises lead to the desired conclusion as follows.

Step	Reason
1. $\neg p \wedge q$	Premise
2. $\neg p$	Simplification using (1)
3. $r \rightarrow p$	Premise
4. $\neg r$	Modus tollens using (2) and (3)
5. $\neg r \rightarrow s$	Premise
6. $s$	Modus ponens using (4) and (5)
7. $s \rightarrow t$	Premise
8. $t$	Modus ponens using (6) and (7)

Note that we could have used a truth table to show that whenever each of the four hypotheses is true, the conclusion is also true. However, because we are working with five propositional variables,  $p$ ,  $q$ ,  $r$ ,  $s$ , and  $t$ , such a truth table would have 32 rows. 

**EXAMPLE 7** Show that the premises “If you send me an e-mail message, then I will finish writing the program,” “If you do not send me an e-mail message, then I will go to sleep early,” and “If I go to sleep early, then I will wake up feeling refreshed” lead to the conclusion “If I do not finish writing the program, then I will wake up feeling refreshed.”

**Solution:** Let  $p$  be the proposition “You send me an e-mail message,”  $q$  the proposition “I will finish writing the program,”  $r$  the proposition “I will go to sleep early,” and  $s$  the proposition “I will wake up feeling refreshed.” Then the premises are  $p \rightarrow q$ ,  $\neg p \rightarrow r$ , and  $r \rightarrow s$ . The desired conclusion is  $\neg q \rightarrow s$ . We need to give a valid argument with premises  $p \rightarrow q$ ,  $\neg p \rightarrow r$ , and  $r \rightarrow s$  and conclusion  $\neg q \rightarrow s$ .

This argument form shows that the premises lead to the desired conclusion.

Step	Reason
1. $p \rightarrow q$	Premise
2. $\neg q \rightarrow \neg p$	Contrapositive of (1)
3. $\neg p \rightarrow r$	Premise
4. $\neg q \rightarrow r$	Hypothetical syllogism using (2) and (3)
5. $r \rightarrow s$	Premise
6. $\neg q \rightarrow s$	Hypothetical syllogism using (4) and (5)

## Resolution

Computer programs have been developed to automate the task of reasoning and proving theorems. Many of these programs make use of a rule of inference known as **resolution**. This rule of inference is based on the tautology



$$((p \vee q) \wedge (\neg p \vee r)) \rightarrow (q \vee r).$$

(Exercise 30 in Section 1.3 asks for the verification that this is a tautology.) The final disjunction in the resolution rule,  $q \vee r$ , is called the **resolvent**. When we let  $q = r$  in this tautology, we obtain  $(p \vee q) \wedge (\neg p \vee q) \rightarrow q$ . Furthermore, when we let  $r = F$ , we obtain  $(p \vee q) \wedge (\neg p) \rightarrow q$  (because  $q \vee F \equiv q$ ), which is the tautology on which the rule of disjunctive syllogism is based.

**EXAMPLE 8** Use resolution to show that the hypotheses “Jasmine is skiing or it is not snowing” and “It is snowing or Bart is playing hockey” imply that “Jasmine is skiing or Bart is playing hockey.”



**Solution:** Let  $p$  be the proposition “It is snowing,”  $q$  the proposition “Jasmine is skiing,” and  $r$  the proposition “Bart is playing hockey.” We can represent the hypotheses as  $\neg p \vee q$  and  $p \vee r$ , respectively. Using resolution, the proposition  $q \vee r$ , “Jasmine is skiing or Bart is playing hockey,” follows.

Resolution plays an important role in programming languages based on the rules of logic, such as Prolog (where resolution rules for quantified statements are applied). Furthermore, it can be used to build automatic theorem proving systems. To construct proofs in propositional logic using resolution as the only rule of inference, the hypotheses and the conclusion must be expressed as **clauses**, where a clause is a disjunction of variables or negations of these variables. We can replace a statement in propositional logic that is not a clause by one or more equivalent statements that are clauses. For example, suppose we have a statement of the form  $p \vee (q \wedge r)$ . Because  $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ , we can replace the single statement  $p \vee (q \wedge r)$  by two statements  $p \vee q$  and  $p \vee r$ , each of which is a clause. We can replace a statement of the form  $\neg(p \vee q)$  by the two statements  $\neg p$  and  $\neg q$  because De Morgan’s law tells us that  $\neg(p \vee q) \equiv \neg p \wedge \neg q$ . We can also replace a conditional statement  $p \rightarrow q$  with the equivalent disjunction  $\neg p \vee q$ .

## Exercises

---

1. Find the argument form for the following argument and determine whether it is valid. Can we conclude that the conclusion is true if the premises are true?

If Socrates is human, then Socrates is mortal.  
Socrates is human.  
\_\_\_\_\_  
∴ Socrates is mortal.

2. Find the argument form for the following argument and determine whether it is valid. Can we conclude that the conclusion is true if the premises are true?

If George does not have eight legs, then he is not a spider.  
George is a spider.  
\_\_\_\_\_  
∴ George has eight legs.

3. What rule of inference is used in each of these arguments?

- a) Alice is a mathematics major. Therefore, Alice is either a mathematics major or a computer science major.
  - b) Jerry is a mathematics major and a computer science major. Therefore, Jerry is a mathematics major.
  - c) If it is rainy, then the pool will be closed. It is rainy. Therefore, the pool is closed.
  - d) If it snows today, the university will close. The university is not closed today. Therefore, it did not snow today.
  - e) If I go swimming, then I will stay in the sun too long. If I stay in the sun too long, then I will sunburn. Therefore, if I go swimming, then I will sunburn.
4. What rule of inference is used in each of these arguments?
- a) Kangaroos live in Australia and are marsupials. Therefore, kangaroos are marsupials.
  - b) It is either hotter than 100 degrees today or the pollution is dangerous. It is less than 100 degrees outside today. Therefore, the pollution is dangerous.
  - c) Linda is an excellent swimmer. If Linda is an excellent swimmer, then she can work as a lifeguard. Therefore, Linda can work as a lifeguard.
  - d) Steve will work at a computer company this summer. Therefore, this summer Steve will work at a computer company or he will be a beach bum.

- e) If I work all night on this homework, then I can answer all the exercises. If I answer all the exercises, I will understand the material. Therefore, if I work all night on this homework, then I will understand the material.
- 5. Use rules of inference to show that the hypotheses "Randy works hard," "If Randy works hard, then he is a dull boy," and "If Randy is a dull boy, then he will not get the job" imply the conclusion "Randy will not get the job."
- 6. Use rules of inference to show that the hypotheses "If it does not rain or if it is not foggy, then the sailing race will be held and the lifesaving demonstration will go on," "If the sailing race is held, then the trophy will be awarded," and "The trophy was not awarded" imply the conclusion "It rained."
- 7. What rules of inference are used in this famous argument? "All men are mortal. Socrates is a man. Therefore, Socrates is mortal."
- 8. What rules of inference are used in this argument? "No man is an island. Manhattan is an island. Therefore, Manhattan is not a man."
- 9. For each of these collections of premises, what relevant conclusion or conclusions can be drawn? Explain the rules of inference used to obtain each conclusion from the premises.
  - a) "If I take the day off, it either rains or snows." "I took Tuesday off or I took Thursday off." "It was sunny on Tuesday." "It did not snow on Thursday."
  - b) "If I eat spicy foods, then I have strange dreams." "I have strange dreams if there is thunder while I sleep." "I did not have strange dreams."
  - c) "I am either clever or lucky." "I am not lucky." "If I am lucky, then I will win the lottery."
  - d) "Every computer science major has a personal computer." "Ralph does not have a personal computer." "Ann has a personal computer."
  - e) "What is good for corporations is good for the United States." "What is good for the United States is good for you." "What is good for corporations is for you to buy lots of stuff!"
  - f) "All rodents gnaw their food." "Mice are rodents." "Rabbits do not gnaw their food." "Bats are not rodents."

- 10.** For each of these sets of premises, what relevant conclusion or conclusions can be drawn? Explain the rules of inference used to obtain each conclusion from the premises.
- "If I play hockey, then I am sore the next day." "I use the whirlpool if I am sore." "I did not use the whirlpool."
  - "If I work, it is either sunny or partly sunny." "I worked last Monday or I worked last Friday." "It was not sunny on Tuesday." "It was not partly sunny on Friday."
  - "All insects have six legs." "Dragonflies are insects." "Spiders do not have six legs." "Spiders eat dragonflies."
  - "Every student has an Internet account." "Homer does not have an Internet account." "Maggie has an Internet account."
  - "All foods that are healthy to eat do not taste good." "Tofu is healthy to eat." "You only eat what tastes good." "You do not eat tofu." "Cheeseburgers are not healthy to eat."
  - "I am either dreaming or hallucinating." "I am not dreaming." "If I am hallucinating, I see elephants running down the road."
- 11.** Show that the argument form with premises  $p_1, p_2, \dots, p_n$  and conclusion  $q \rightarrow r$  is valid if the argument form with premises  $p_1, p_2, \dots, p_n, q$ , and conclusion  $r$  is valid.
- 12.** Show that the argument form with premises  $(p \wedge t) \rightarrow (r \vee s)$ ,  $q \rightarrow (u \wedge t)$ ,  $u \rightarrow p$ , and  $\neg s$  and conclusion  $q \rightarrow r$  is valid by first using Exercise 11 and then using rules of inference from Table 1.
- 13.** For each of these arguments, explain which rules of inference are used for each step.
- "Doug, a student in this class, knows how to write programs in JAVA. Everyone who knows how to write programs in JAVA can get a high-paying job. Therefore, someone in this class can get a high-paying job."
  - "Somebody in this class enjoys whale watching. Every person who enjoys whale watching cares about ocean pollution. Therefore, there is a person in this class who cares about ocean pollution."
  - "Each of the 93 students in this class owns a personal computer. Everyone who owns a personal computer can use a word processing program. Therefore, Zeke, a student in this class, can use a word processing program."
  - "Everyone in New Jersey lives within 50 miles of the ocean. Someone in New Jersey has never seen the ocean. Therefore, someone who lives within 50 miles of the ocean has never seen the ocean."
- 14.** For each of these arguments, explain which rules of inference are used for each step.
- "Linda, a student in this class, owns a red convertible. Everyone who owns a red convertible has gotten at least one speeding ticket. Therefore, someone in this class has gotten a speeding ticket."
  - "Each of five roommates, Melissa, Aaron, Ralph, Veneesha, and Keeshawn, has taken a course in discrete mathematics. Every student who has taken a course in discrete mathematics can take a course in algorithms. Therefore, all five roommates can take a course in algorithms next year."
  - "All movies produced by John Sayles are wonderful. John Sayles produced a movie about coal miners. Therefore, there is a wonderful movie about coal miners."
  - "There is someone in this class who has been to France. Everyone who goes to France visits the Louvre. Therefore, someone in this class has visited the Louvre."
- 15.** For each of these arguments determine whether the argument is correct or incorrect and explain why.
- All students in this class understand logic. Xavier is a student in this class. Therefore, Xavier understands logic.
  - Every computer science major takes discrete mathematics. Natasha is taking discrete mathematics. Therefore, Natasha is a computer science major.
  - All parrots like fruit. My pet bird is not a parrot. Therefore, my pet bird does not like fruit.
  - Everyone who eats granola every day is healthy. Linda is not healthy. Therefore, Linda does not eat granola every day.
- 16.** For each of these arguments determine whether the argument is correct or incorrect and explain why.
- Everyone enrolled in the university has lived in a dormitory. Mia has never lived in a dormitory. Therefore, Mia is not enrolled in the university.
  - A convertible car is fun to drive. Isaac's car is not a convertible. Therefore, Isaac's car is not fun to drive.
  - Quincy likes all action movies. Quincy likes the movie *Eight Men Out*. Therefore, *Eight Men Out* is an action movie.
  - All lobstermen set at least a dozen traps. Hamilton is a lobsterman. Therefore, Hamilton sets at least a dozen traps.
- 17.** What is wrong with this argument? Let  $H(x)$  be " $x$  is happy." Given the premise  $\exists x H(x)$ , we conclude that  $H(\text{Lola})$ . Therefore, Lola is happy.
- 18.** What is wrong with this argument? Let  $S(x, y)$  be " $x$  is shorter than  $y$ ." Given the premise  $\exists s S(s, \text{Max})$ , it follows that  $S(\text{Max}, \text{Max})$ . Then by existential generalization it follows that  $\exists x S(x, x)$ , so that someone is shorter than himself.
- 19.** Determine whether each of these arguments is valid. If an argument is correct, what rule of inference is being used? If it is not, what logical error occurs?
- If  $n$  is a real number such that  $n > 1$ , then  $n^2 > 1$ . Suppose that  $n^2 > 1$ . Then  $n > 1$ .
  - If  $n$  is a real number with  $n > 3$ , then  $n^2 > 9$ . Suppose that  $n^2 \leq 9$ . Then  $n \leq 3$ .
  - If  $n$  is a real number with  $n > 2$ , then  $n^2 > 4$ . Suppose that  $n \leq 2$ . Then  $n^2 \leq 4$ .

**Extra Examples**

to show that  $n$  is odd? We see that  $3n + 1 = 2k$ , but there does not seem to be any direct way to conclude that  $n$  is odd. Because our attempt at a direct proof failed, we next try a proof by contraposition.

The first step in a proof by contraposition is to assume that the conclusion of the conditional statement “If  $3n + 2$  is odd, then  $n$  is odd” is false; namely, assume that  $n$  is even. Then, by the definition of an even integer,  $n = 2k$  for some integer  $k$ . Substituting  $2k$  for  $n$ , we find that  $3n + 2 = 3(2k) + 2 = 6k + 2 = 2(3k + 1)$ . This tells us that  $3n + 2$  is even (because it is a multiple of 2), and therefore not odd. This is the negation of the premise of the theorem. Because the negation of the conclusion of the conditional statement implies that the hypothesis is false, the original conditional statement is true. Our proof by contraposition succeeded; we have proved the theorem “If  $3n + 2$  is odd, then  $n$  is odd.”

**EXAMPLE 4** Prove that if  $n = ab$ , where  $a$  and  $b$  are positive integers, then  $a \leq \sqrt{n}$  or  $b \leq \sqrt{n}$ .

**Solution:** Because there is no obvious way of showing that  $a \leq \sqrt{n}$  or  $b \leq \sqrt{n}$  directly from the equation  $n = ab$ , where  $a$  and  $b$  are positive integers, we attempt a proof by contraposition.

The first step in a proof by contraposition is to assume that the conclusion of the conditional statement “If  $n = ab$ , where  $a$  and  $b$  are positive integers, then  $a \leq \sqrt{n}$  or  $b \leq \sqrt{n}$ ” is false. That is, we assume that the statement  $(a \leq \sqrt{n}) \vee (b \leq \sqrt{n})$  is false. Using the meaning of disjunction together with De Morgan’s law, we see that this implies that both  $a \leq \sqrt{n}$  and  $b \leq \sqrt{n}$  are false. This implies that  $a > \sqrt{n}$  and  $b > \sqrt{n}$ . We can multiply these inequalities together (using the fact that if  $0 < s < t$  and  $0 < u < v$ , then  $su < tv$ ) to obtain  $ab > \sqrt{n} \cdot \sqrt{n} = n$ . This shows that  $ab \neq n$ , which contradicts the statement  $n = ab$ .

Because the negation of the conclusion of the conditional statement implies that the hypothesis is false, the original conditional statement is true. Our proof by contraposition succeeded; we have proved that if  $n = ab$ , where  $a$  and  $b$  are positive integers, then  $a \leq \sqrt{n}$  or  $b \leq \sqrt{n}$ .

**VACUOUS AND TRIVIAL PROOFS** We can quickly prove that a conditional statement  $p \rightarrow q$  is true when we know that  $p$  is false, because  $p \rightarrow q$  must be true when  $p$  is false. Consequently, if we can show that  $p$  is false, then we have a proof, called a **vacuous proof**, of the conditional statement  $p \rightarrow q$ . Vacuous proofs are often used to establish special cases of theorems that state that a conditional statement is true for all positive integers [i.e., a theorem of the kind  $\forall n P(n)$ , where  $P(n)$  is a propositional function]. Proof techniques for theorems of this kind will be discussed in Section 5.1.

**EXAMPLE 5** Show that the proposition  $P(0)$  is true, where  $P(n)$  is “If  $n > 1$ , then  $n^2 > n$ ” and the domain consists of all integers.

**Solution:** Note that  $P(0)$  is “If  $0 > 1$ , then  $0^2 > 0$ .” We can show  $P(0)$  using a vacuous proof. Indeed, the hypothesis  $0 > 1$  is false. This tells us that  $P(0)$  is automatically true.

**Remark:** The fact that the conclusion of this conditional statement,  $0^2 > 0$ , is false is irrelevant to the truth value of the conditional statement, because a conditional statement with a false hypothesis is guaranteed to be true.

We can also quickly prove a conditional statement  $p \rightarrow q$  if we know that the conclusion  $q$  is true. By showing that  $q$  is true, it follows that  $p \rightarrow q$  must also be true. A proof of  $p \rightarrow q$  that uses the fact that  $q$  is true is called a **trivial proof**. Trivial proofs are often important when special cases of theorems are proved (see the discussion of proof by cases in Section 1.8) and in mathematical induction, which is a proof technique discussed in Section 5.1.

# **Chapter (3)**

## **Recursion and Counting**

## The Basic of Counting

### Introduction:

---

- Suppose that a password on a computer system consists of **eight** characters.
- Each of these characters must be a **digit** or a **letter** of the alphabet.
- Each password must contain at **least one digit**.
- **How many such passwords are there??**



## Product Rule

### Multiplication (Product) Rule:

- Suppose that a **procedure** can be broken down into a sequence of two tasks.
- If there are  $n_1$  ways to do the first task and for each of these ways of doing the first task, there are  $n_2$  ways to do the second task,
- then there are  $n_1 \times n_2$  ways to do the **procedure**.

## Multiplication (Product) Rule

- Suppose that a **procedure** can be broken down into a sequence of n tasks.
  - The total number of ways to complete the operation is  $n_1 \times n_2 \times \dots \times n_k$
- 

11

### Example (1)

A new company with just two employees, Sanchez and John, rents a floor of a building with 12 offices.

**How many ways are there to assign different offices to these two employees?**

#### Solution:

Sanchez	John

---

12

## Example (1) : Solution

### Solution:

Sanchez	John
First, we have 12 offices Then, we select 1 from 12 offices	Second, we have 11 offices Then, we select 1 from 11 offices
$n_1 = 12$ ways	$n_2 = 11$ ways
<b>Total</b> = $12 \times 11 = 132$ ways to assign offices to these two employees.	

13

## Example (2)

How many different **bit strings** of length **seven** are there?

### Solution:

Bits #	1	2	3	4	5	6	7
Value							
Ways							

14

## Example (2)

How many different **bit strings** of length **seven** are there?

### Solution:

Each of the seven bits can be chosen in two ways, because each bit is either 0 or 1. Therefore, the product rule shows there are a total of  $2^7 = 128$  different bit strings of length seven.

Bits #	1	2	3	4	5	6	7
Value	either 0 or 1						
Ways	$n_1 = 2$	$n_2 = 2$	$n_3 = 2$	$n_4 = 2$	$n_5 = 2$	$n_6 = 2$	$n_7 = 2$
<b>Total</b> = $2^7 = 128$ different bit strings of length seven.							

15

## Example (3)

In how many different ways can a **true-false** test consisting of **10 questions** be answered?

### Solution:

Question #	Value	Ways
1		
2		
3		
10		

16

## Example (3)

---

In how many different ways can a **true-false** test consisting of **10 questions** be answered?

### Solution:

Each of the 10 questions can be chosen in **two ways**, because each question is **either true or false**.

Therefore, the product rule shows there are:

$$2 \times 2 \times \dots \times 2 = 2^{10} = 1024 \text{ ways to answer the test.}$$

---

17

## Example (4)

The design for a Website is to consist of **four colors**, **three fonts**, and **three positions** for an image.

How many different designs are possible?

### Solution:

From the product rule,  $4 \times 3 \times 3 = 36$  different designs are possible.

---

18

## Example (5)

How many bit strings of length 5, start and end with 1's?

### Solution:

Bits #	1	2	3	4	5
Value					
Ways					

---

19

## Example (5)

How many bit strings of length 5, start and end with 1's?

### Solution:

Bits #	1	2	3	4	5
Value	1	either 0 or 1	either 0 or 1	either 0 or 1	1
Ways	$n_1 = 1$	$n_2 = 2$	$n_3 = 2$	$n_4 = 2$	$n_5 = 1$

**Total** =  $1 \times 2 \times 2 \times 2 \times 1 = 8$  different bit strings of length 5, start and end with 1's.

---

20

## Example (6)

How many different license plates are available if each plate contains a sequence of three letters followed by three digits.

**Solution:**

ABC 123

---

21

## Example (6)

How many different license plates are available if each plate contains a sequence of three letters followed by three digits.

**Solution:**

ABC 123

— — —      — — —  
26 choices      10 choices  
for each      for each  
letter      digit

There are 26 choices for each of the three letters and ten choices for each of the three digits. Hence, by the product rule there are a total of  $26 \cdot 26 \cdot 26 \cdot 10 \cdot 10 \cdot 10 = 17,576,000$  possible license plates.

---

22

## The product rule in terms of sets

The product rule is often phrased in terms of sets in this way: If  $A_1, A_2, \dots, A_m$  are finite sets, then the number of elements in the Cartesian product of these sets is the product of the number of elements in each set.

To relate this to the product rule, note that the task of choosing an element in the Cartesian product  $A_1 \times A_2 \times \dots \times A_m$  is done by choosing an element in  $A_1$ , an element in  $A_2, \dots$ , and an element in  $A_m$ . By the product rule it follows that

$$|A_1 \times A_2 \times \dots \times A_m| = |A_1| \cdot |A_2| \cdot \dots \cdot |A_m|$$

23

## The product rule in terms of sets

Let A, B, and C be three sets, where  $A = \{0, 1\}$ ,  $B = \{1, 2\}$ , and  $C = \{0, 1, 2\}$

How many elements in the Cartesian products of the sets of A, B, and C

### Solution:

$$\begin{aligned} A \times B \times C &= \{(0,1,0), (0,1,1), (0,1,2), (0,2,0), (0,2,1), (0,2,2), \\ &\quad (1,1,0), (1,1,1), (1,1,2), (1,2,0), (1,2,1), (1,2,2)\} \end{aligned}$$

$$|A \times B \times C| = |A||B||C| = 2 \cdot 2 \cdot 3 = 12$$

24

## The Sum Rule

### The Sum Rule

---

- If a task can be done **either** in  $n_1$  ways **or** in  $n_2$  ways, where **none of the set** of  $n_1$  ways is **the same** as any of the set of  $n_2$  ways, then there are  $n_1 + n_2$  ways to do the task.
-

## The sum rule in terms of sets :

---

$$|A_1 \cup A_2 \cup \dots \cup A_m| = |A_1| + |A_2| + \dots + |A_m|$$

if  $A_1, A_2, \dots, A_m$  disjoint

if  $A_1, A_2$  NOT disjoint

$$\therefore |A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|$$

---

27

## Example (7)

---

Suppose that **either** a member of the mathematics major **or** a student who is a physics major is chosen as a representative to a university committee. How many different choices are there for this representative if there are **37** members of the mathematics majors and **83** physics majors and **no one is both** a mathematics and a physics major?

### **Solution:**

By the sum rule it follows that there are  $37 + 83 = 120$  possible ways.

---

## Example (8)

---

A student can choose a computer project from one of three lists. The three lists contain 23, 15, and 19 possible projects, respectively. No project is on more than one list. How many possible projects are there to choose from?

### **Solution:**

By the sum rule there are  $23 + 15 + 19 = 57$  ways to choose a project.

---

29

## Example (10)

---

Each user on a computer system has a password, which is **six to eight characters long**, where each character is an uppercase letter or a digit. Each password must contain at **least one digit**.

**How many possible passwords are there?**

### **Solution:**

---

30

## Example (10)

### Solution:

Let  $P$  be the total number of possible passwords, and let  $P_6$ ,  $P_7$ , and  $P_8$  denote the number of possible passwords of length 6, 7, and 8, respectively. By the sum rule,  
$$P = P_6 + P_7 + P_8.$$

31

## Example (10)

Finding  $P_6$  directly is difficult. To find  $P_6$  it is easier to find the number of strings of **uppercase letters and digits** that are **six characters** long, including those with no digits, and **subtract** from this the number of **strings with no digits**.

By the **product** rule, the number of strings of **six characters** is  $36^6$ , and the number of strings with **no digits** is  $26^6$ . Hence,

$$P_6 = 36^6 - 26^6$$

# of uppercase letters = 26  
+  
# of digits = 10

32

## Example (10)

$$P_6 = 36^6 - 26^6 = 2,176,782,336 - 308,915,776 = 1,867,866,560.$$

$$P_7 = 36^7 - 26^7 = 78,364,164,096 - 8,031,810,176 = 70,332,353,920.$$

$$P_8 = 36^8 - 26^8 = 2,821,109,907,456 - 208,827,064,576 = 2,612,282,842,880.$$

By the sum rule,  $P = P_6 + P_7 + P_8$ .

$$P = P_6 + P_7 + P_8$$

$$= 1,867,866,560 + 70,332,353,920 + 2,612,282,842,880$$

$$= 2,684,483,063,360$$

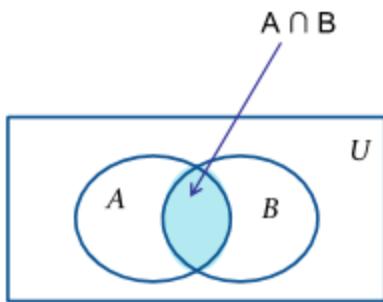
---

33

## The Subtraction Rule

## The Subtraction Rule

If a task can be done either in  $n_1$  ways or in  $n_2$  ways, then the number of ways to do the task is  $n_1 + n_2$  minus the number of ways to do the task that are common to the two different ways.



---

35

## The principle of inclusion–exclusion:

The subtraction rule is also known as the *principle of inclusion–exclusion*, especially when it is used to count the number of elements in the union of two sets.

$$\text{if } A_1, A_2 \text{ NOT disjoint} \\ \therefore |A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|$$

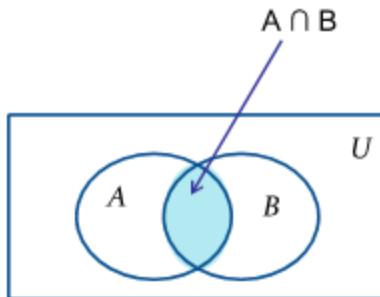
---

36

## The principle of inclusion–exclusion:

- The Cardinality of the Union of Two Sets
- Theorem 2: If A and B are finite sets, then

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



---

37

## Example (11)

How many bit strings of length **five** either start with a 1 bit or end with the two bits 00 ?

### Solution:

1	--	--	--	--
---	----	----	----	----

or

--	--	--	0	0
----	----	----	---	---

1	0	1	0	1
---	---	---	---	---

1	0	0	0	0
---	---	---	---	---

1	1	1	0	0
---	---	---	---	---

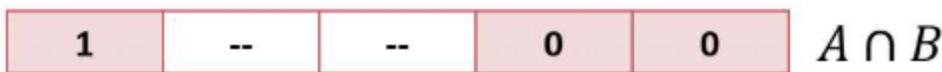
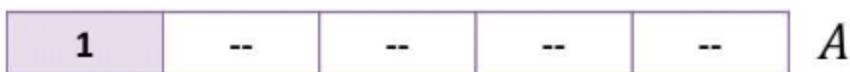
1	1	1	0	0
---	---	---	---	---

1	0	0	0	0
---	---	---	---	---

0	1	1	0	0
---	---	---	---	---

## Example (11)

**Solution:**



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

---

39

## Example (11)

**Solution:**



Bits #	1	2	3	4	5
Value	1	either 0 or 1	either 0 or 1	either 0 or 1	either 0 or 1
Ways	$n_1 = 1$	$n_2 = 2$	$n_3 = 2$	$n_4 = 2$	$n_5 = 2$

Total =  $1 \times 2 \times 2 \times 2 \times 2 = 2^4$  different bit strings of length 5, start with 1.

---

40

## Example (11)

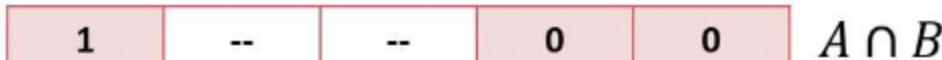


Bits #	1	2	3	4	5
Value	either 0 or 1	either 0 or 1	either 0 or 1	0	0
Ways	$n_1 = 2$	$n_2 = 2$	$n_3 = 2$	$n_4 = 1$	$n_5 = 1$

**Total** =  $2 \times 2 \times 2 \times 1 \times 1 = 2^3$  different bit strings of length 5, end with the two bits 00.

41

## Example (11)



Bits #	1	2	3	4	5
Value	1	either 0 or 1	either 0 or 1	0	0
Ways	$n_1 = 1$	$n_2 = 2$	$n_3 = 2$	$n_4 = 1$	$n_5 = 1$

**Total** =  $1 \times 2 \times 2 \times 1 \times 1 = 2^2$  different bit strings of length 5, start with a 1 bit AND end with the two bits 00.

42

## Example (11)

---

The total number of bit strings of length five either start with a 1 bit or end with the two bits 00 is:

$$= 2^4 + 2^3 - 2^2 = 16 + 8 - 4 = 20$$

---

43

## Example (12)

---

A computer company receives 350 applications from college graduates for a job planning a line of new web servers. Suppose that 220 of these applicants majored in computer science, 147 majored in business, and 51 majored both in computer science and in business. **How many of these applicants majored neither in computer science nor in business?**

---

44

## Example (12)

### Solution:

220 computer science  $|A|$

147 business  $|B|$

51 both  $|A \cap B|$

Total 350 applications  $|U|$

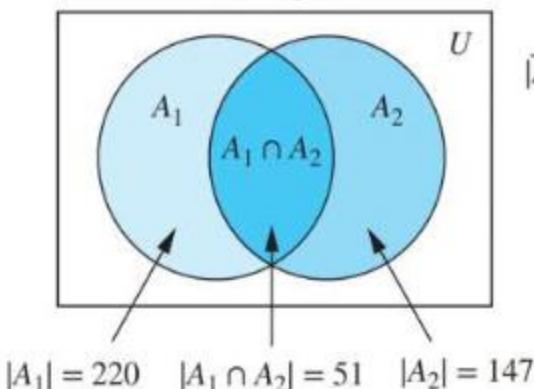
The applicants majored **neither** in computer science **nor** in business

$$\begin{aligned} &= |U| - |A \cup B| \\ &= |U| - (|A| + |B| - |A \cap B|) = 350 - (220 + 147 - 51) = 34 \end{aligned}$$

45

## Example (12)

Venn diagram



$$\begin{aligned} |\overline{A_1 \cup A_2}| &= |U| - |A_1 \cup A_2| \\ &= |U| - (|A_1| + |A_2| - |A_1 \cap A_2|) \\ &= 350 - (220 + 147 - 51) \\ &= 350 - 316 \\ &= 34 \end{aligned}$$

46

## Permutations and Combinations

### Introduction:

---

- Many **counting problems** can be solved by finding the number of ways to **arrange** a specified number of **distinct elements** of a set of a particular size, where the **order** of these elements' **matters**.
  - Many other counting problems can be solved by finding the number of ways to select a particular number of elements from a set of a particular size, **where the order** of the elements selected **does not matter**.
-

## Permutations

### Permutation

- A permutation of a set of **distinct objects** is an **ordered** arrangement of these objects. For instant, find the number of **ordered** sequences of the elements of a set. Consider a set of elements, such as  $S = \{a, b, c\}$ .
- A permutation of the elements is an ordered sequence of the elements. For example,  $abc$ ,  $acb$ ,  $bac$ ,  $bca$ ,  $cab$ , and  $cba$  are all of the permutations of the elements of  $S$ .

$$3 \times 2 \times 1 = 6$$

## Permutation

- The number of permutations of  $n$  different elements is  **$n!$**  where

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$$

- For instance, the number of permutations of the four letters  **$a, b, c,$  and  $d$**  will be  $4! = 24$ .

---

51

## Example (13)

How many permutations of the letters A, B, C, D, E, F, G, H contain the string **ABC** ?

### Solution:

Because the letters ABC must occur as a block, we can find the answer by finding the number of permutations of six objects, namely, the block ABC and the individual letters D, E, F, G, and H. Because these six objects can occur in any order, there are  $6! = 720$  permutations of the letters ABCDEFGH in which ABC occurs as a block.

---

52

## **r-Permutation**

We also are interested in **ordered** arrangements of some of the elements of a set. An **ordered** arrangement of  $r$  elements of a set is called **an  $r$ -permutation**.

The number of **permutations** of subsets of  $r$  elements selected from a set of  $n$  different elements is

$$P(n, r) = P_r^n = {}_n P_r \\ = n \times (n - 1) \times (n - 2) \times \cdots \times (n - r + 1) = \frac{n!}{(n - r)!}$$

where  $1 \leq r \leq n$

---

53

### **Example (14)**

Consider a set of elements, such as  $S = \{a, b, c, d, e\}$ .

What is the number of permutation of **subsets** of 3 elements selected from  $S$  is?

#### **Solution:**

$$r = 3, \quad n = 5$$

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

$$P(5, 3) = \frac{5!}{(5 - 3)!}$$

$$= \frac{5!}{(2)!} = \frac{5 \times 4 \times 3 \times 2 \times 1}{2 \times 1} = 60$$

---

54

## Example (15)

How many ways are there to select a first-prize winner, a second-prize winner, and a third-prize winner from 100 different people who have entered a contest?

**Solution:**

$$r = 3, \quad n = 100$$

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

$$\begin{aligned} P(100, 3) &= \frac{100!}{(100 - 3)!} \\ &= \frac{100!}{(97)!} = 100 \times 99 \times 98 = 970,200 \end{aligned}$$

---

55

**Combinations**

## Combinations

We now turn our attention to counting **unordered** selections of objects. To find the number of subsets of a particular size of a set with  $n$  elements, where  $n$  is a positive integer. An  **$r$ -combination** of elements of a set is an **unordered** selection of  $r$  elements from the set.

$$C(n, r) = C_r^n = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

$$C(n, r) = C(n, n-r)$$

57

### Example (16)

How many possible selections of 3 balls from a box contains 10 colored balls ?

**Solution:**

$$r = 3, \quad n = 10$$

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

$$C(10, 3) = \frac{10!}{3!(10-3)!}$$

$$= \frac{10!}{3!7!} = 120$$

58

## Example (17)

How many ways are there to select five players from a 10 member tennis team to make a trip to a match at another school ?

### Solution:

$$r = 5, \quad n = 10$$

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

$$\begin{aligned} C(10, 5) &= \frac{10!}{5!(10-5)!} \\ &= \frac{10!}{5!5!} = 252 \end{aligned}$$

---

59

## Example (18)

Suppose that there are 9 faculty members in the mathematics department and 11 in the computer science department. How many ways are there to select a committee to develop a **discrete mathematics** course at a school if the committee is to consist of **three faculty** members from the **mathematics** department and **four from** the **computer science** department?

### Solution:

$$C(9, 3) \cdot C(11, 4) = \frac{9!}{3!6!} \cdot \frac{11!}{4!7!} = 84 \cdot 330 = 27,720$$

---

60

## Example (19)

How many bit strings of length 10 contain exactly four 1s?

### Solution:

Location	1	2	3	4	5	6	7	8	9	10
Bit										

Location	1	2	3	4	5	6	7	8	9	10
Bit	1	1	1	1	0	0	0	0	0	0

Location	1	2	3	4	5	6	7	8	9	10
Bit	0	0	0	1	1	1	1	0	0	0

This is just asking us to choose 4 out of 10 slots to place 1's in.

$$C(10, 4) = \frac{10!}{4! \times 6!} = 210$$

61

## Example (20)

How many bit strings of length  $n$  contain exactly  $r$  1s?

### Solution:

Location	1	2	3	4	...	$n-2$	$n-1$	$n$
Bit								

Location	1	2	3	4	...	$n-2$	$n-1$	$n$		
Bit	1	1	0	1		...	0	1	0	1

This is just asking us to choose  $r$  out of  $n$  slots to place 1's in.

The positions of  $r$  1s in a bit string of length  $n$  form an  $r$ -combination of the set  $\{1, 2, 3, \dots, n\}$ . Hence, there are

$C(n, r)$  bit strings of length  $n$  that contain exactly  $r$  1s.

62

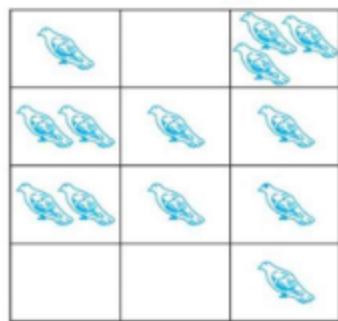
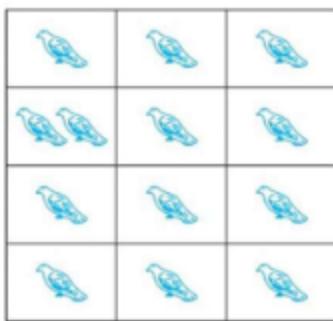
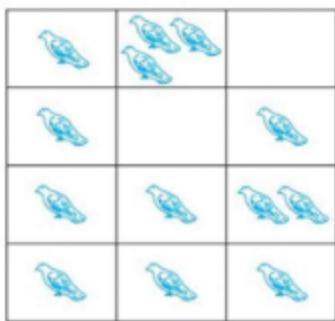
## The Pigeonhole Principle

### Introduction:

Suppose that a flock (قطيع) of 20 pigeons flies into a set of 19 pigeonholes to roost. Because there are 20 pigeons but only 19 pigeonholes, at least one of these 19 pigeonholes must have **at least two pigeons** in it. To see why this is true, note that if each pigeonhole had at most one pigeon in it, at most 19 pigeons, one per hole, could be accommodated.

## The Pigeonhole Principle

This illustrates a general principle called the **pigeonhole principle**. For instance, suppose that a flock of 13 pigeons flies into a set of 12 pigeonholes to roost.



## The Pigeonhole Principle:

If  $k$  is a positive integer and  $k + 1$  or more objects are placed into  $k$  boxes, then there is at least one box containing two or more of the objects.

## COROLLARY 1:

---

A function  $f$  from a set with  $k + 1$  or more elements to a set with  $k$  elements is **not** one-to-one.

---

67

## Example (21)

---

Among any group of 367 people, there must be at least two with the **same birthday**, because there are only 366 possible birthdays.

---

68

## Example (22)

---

In any group of **27 English words**, there must be at least two that begin with **the same letter**, because there are 26 letters in the English alphabet.

---

69

## Example (23)

---

How many students must be in a class to guarantee that **at least two** students receive the **same score** on the final exam, if the exam is graded on a scale **from 0 to 100** points?

### **Solution:**

There are 101 possible scores on the final. The pigeonhole principle shows that among any 102 students there must be at least 2 students with the same score.

---

## The Generalized Pigeonhole Principle

The pigeonhole principle states that there must be at least two objects in the same box when there are **more objects** than boxes. However, even more can be said when the number of objects **exceeds a multiple** of the number of boxes.

---

71

## The Generalized Pigeonhole Principle

For instance, suppose that a flock of 25 pigeons flies into a set of 12 pigeonholes to roost.

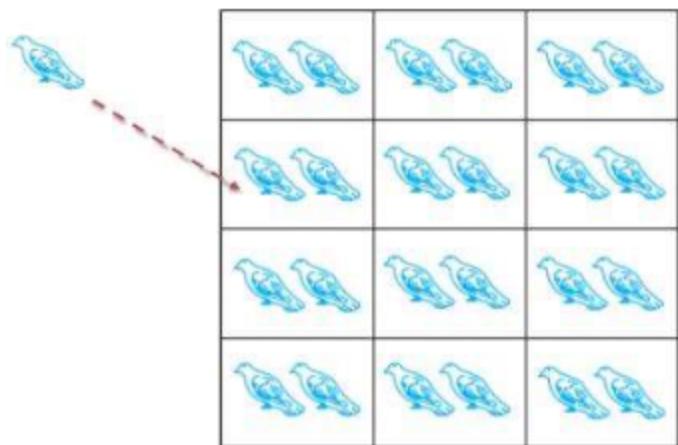


(empty)	(empty)	(empty)

---

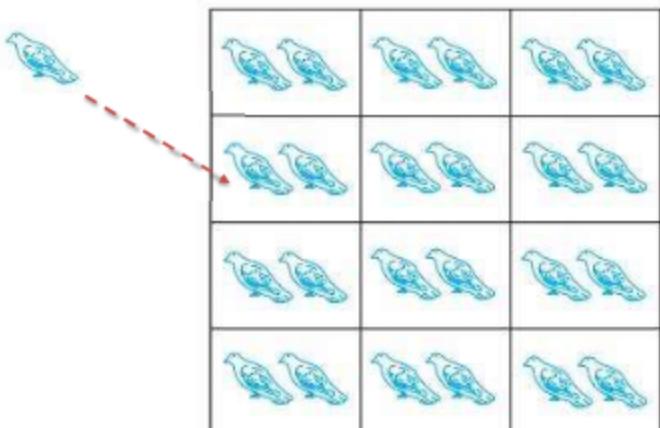
72

## The Generalized Pigeonhole Principle



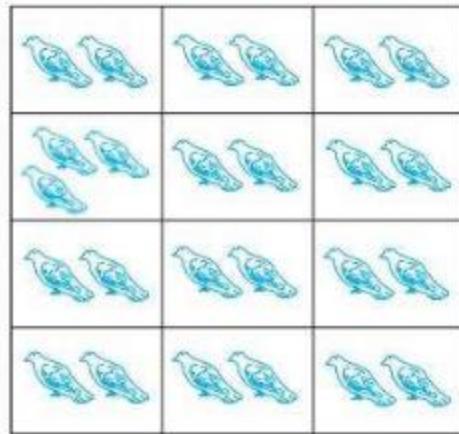
73

## The Generalized Pigeonhole Principle



74

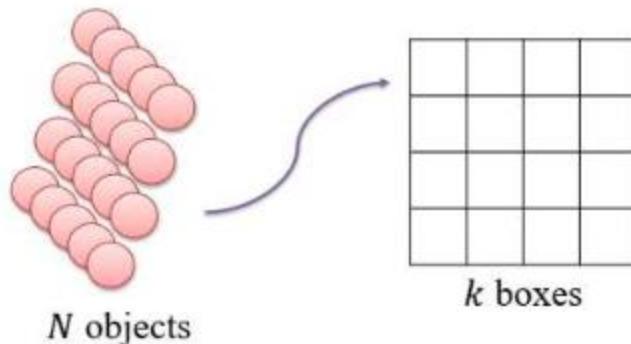
## The Generalized Pigeonhole Principle



75

## The Generalized Pigeonhole Principle

If  $N$  objects are placed into  $k$  boxes, then there is at least one box containing at least  $\lceil N/k \rceil$  objects. Where  $N$  and  $k$  are positive integers.

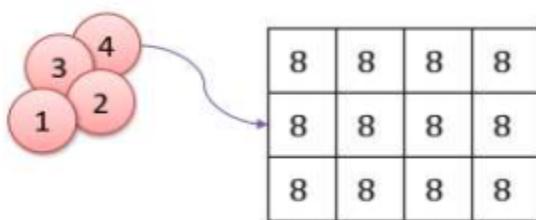


76

## Example (24)

Among 100 people there are **at least**  $[100/12] = 9$  who were born in the same month.

### Solution:



$$N = 100$$

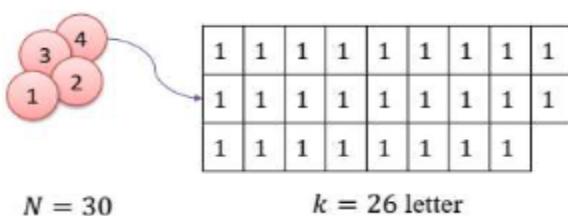
$$k = 12 \\ \text{month}$$

77

## Example (25)

Show that if there are 30 students in a class, then at least two have last names that begin with the same letter.

### Solution:



$$N = 30$$

$$k = 26 \text{ letter}$$

**at least**  $[30/26] = 2$  names that begin with the same letter.

78

### **Section No. 07**

**Answer the following Questions.**

## **The Basics of Counting**

1.	There are 18 mathematics majors and 325 computer science majors at a college.  a) In how many ways can two representatives be picked so that one is a mathematics major and the other is a computer science major?  b) In how many ways can one representative be picked who is either a mathematics major or a computer science major?	
2.	An office building contains 27 floors and has 37 offices on each floor. How many offices are in the building?	
3.	A multiple-choice test contains 10 questions. There are four possible answers for each question.  a) In how many ways can a student answer the questions on the test if the student answers every question?  b) In how many ways can a student answer the questions on the test if the student can leave answers blank?	
4.	A particular brand of shirt comes in 12 colors, has a male version and a female version, and comes in three sizes for each sex. How many different types of this shirt are made?	
5.	Six different airlines fly from New York to Denver and seven fly from Denver to San Francisco. How many different pairs of airlines can you choose on which to book a trip from New York to San Francisco via Denver, when you pick an airline for the flight to Denver and an airline for the continuation flight to San Francisco?	
6.	How many bit strings are there of length eight?	
7.	How many bit strings are there of length six or less, not counting the empty string?	

8.	. How many bit strings of length $n$ , where $n$ is a positive integer, start and end with 1s?	
9.	How many strings of three decimal digits a) do not contain the same digit three times? b) begin with an odd digit? c) have exactly two digits that are 4s?	
10.	How many license plates can be made using either three digits followed by three uppercase English letters or three uppercase English letters followed by three digits?	
11.	How many license plates can be made using either three uppercase English letters followed by three digits or four uppercase English letters followed by two digits?	
12.	. How many bit strings of length seven either begin with two 0s or end with three 1s?	
13.	Every student in a discrete mathematics class is either a computer science or a mathematics major or is a joint major in these two subjects. How many students are in the class if there are 38 computer science majors (including joint majors), 23 mathematics majors (including joint majors), and 7 joint majors?	

## The Pigeonhole Principle

14.	Show that if there are 30 students in a class, then at least two have last names that begin with the same letter.	
15.	A bowl contains 10 red balls and 10 blue balls. A woman selects balls at random without looking at them. a) How many balls must she select to be sure of having at least three balls of the same color? b) How many balls must she select to be sure of having at least three blue balls?	
16.	How many numbers must be selected from the set {1, 2, 3, 4, 5, 6} to guarantee that at least one pair of these numbers add up to 7?	
17.	. How many numbers must be selected from the set {1, 3, 5, 7, 9, 11, 13, 15} to guarantee that at least one pair of these numbers add up to 16?	

## Permutations and Combinations

18.	List all the permutations of $\{a, b, c\}$ .	
19.	How many different permutations are there of the set $\{a, b, c, d, e, f, g\}$ ?	
20.	How many permutations of $\{a, b, c, d, e, f, g\}$ end with $a$ ?	
21.	Let $S = \{1, 2, 3, 4, 5\}$ . a) List all the 3-permutations of $S$ . b) List all the 3-combinations of $S$ .	
22.	Find the value of each of these quantities. a) $P(6, 3)$ b) $P(6, 5)$ c) $P(8, 1)$ d) $P(8, 5)$ e) $P(8, 8)$ f) $P(10, 9)$	
23.	Find the value of each of these quantities. a) $C(5, 1)$ b) $C(5, 3)$ c) $C(8, 4)$ d) $C(8, 8)$ e) $C(8, 0)$ f) $C(12, 6)$	
24.	Find the number of 5-permutations of a set with nine elements.	
25.	In how many different orders can five runners finish a race if no ties are allowed?	
26.	How many possibilities are there for the win, place, and show (first, second, and third) positions in a horse race with 12 horses if all orders of finish are possible?	
27.	There are six different candidates for governor of a state. In how many different orders can the names of the candidates be printed on a ballot?	
28.	How many bit strings of length 10 contain a) exactly four 1s? b) at most four 1s? c) at least four 1s? d) an equal number of 0s and 1s?	

29.

- How many permutations of the letters  $ABCDEFG$  contain
  - a) the string  $BCD$ ?
  - b) the string  $CFG A$ ?
  - c) the strings  $BA$  and  $GF$ ?
  - d) the strings  $ABC$  and  $DE$ ?
  - e) the strings  $ABC$  and  $CDE$ ?
  - f) the strings  $CBA$  and  $BED$ ?

# **Chapter (4)**

## **Graph Theory**

# Chapter 1: Graphs Part(I)

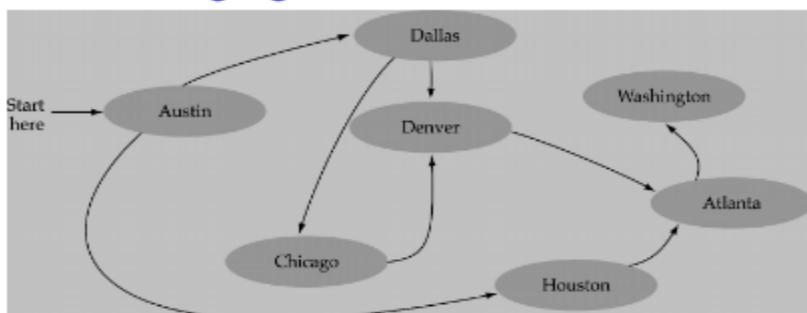
$$G = (V, E)$$

## Introduction of Graphs

A graph consists of a nonempty set **V** of nodes (**vertices**) and a set of edges **E** that relate the nodes to each other

The set of edges describes relationships among the vertices

Ex. An airline files only between the cities connected by lines in following fig.

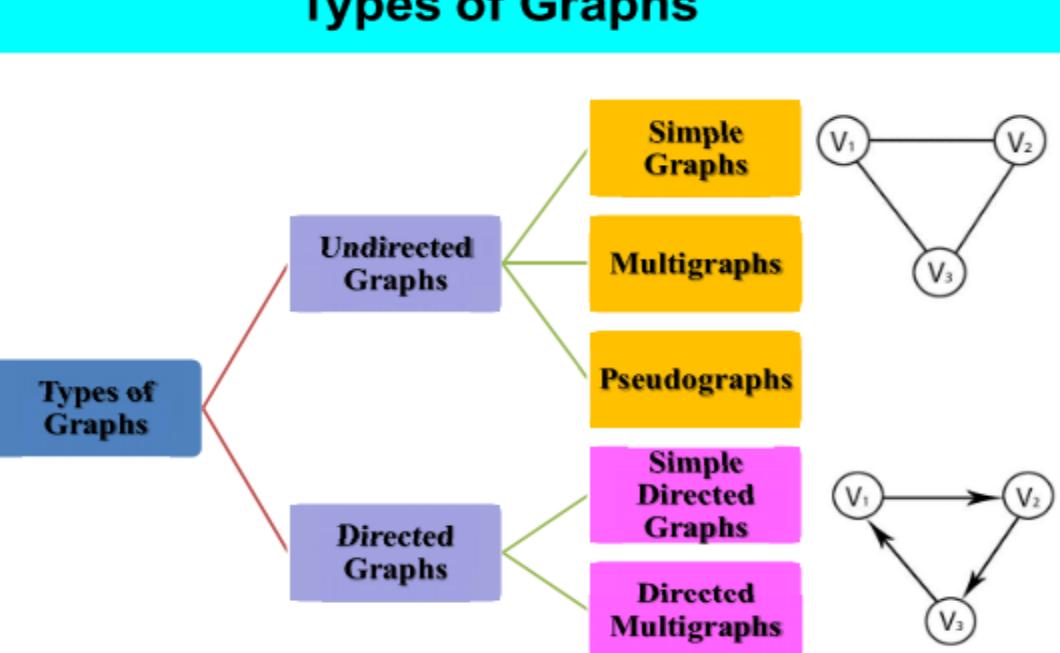


$$V(G) = \{\text{Austin}, \text{Dallas}, \text{Denver}, \text{Washington}, \text{Chicago}, \text{Houston}, \text{Atlanta}\}$$

$$E(G) = \{(\text{Austin}, \text{Dallas}), (\text{Dallas}, \text{Denver}), (\text{Dallas}, \text{Chicago}), \dots\}$$

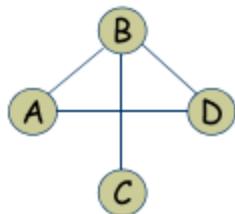
$$G = (V, E) \quad |V| = 7, |E| = 8,$$

## Types of Graphs



## Undirected Graph

- When the edges in a graph have no direction, the graph is called *undirected*.
  - A graph  $G = (V, E)$  consists of
    - V:** a nonempty set of vertices (nodes)
    - E:** a set of edges (unordered pairs of distinct elements of V)



$$G=(V, E)$$

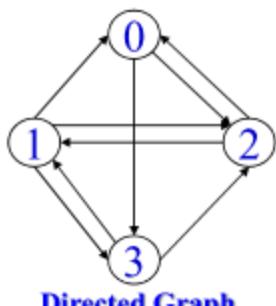
$$V(G)=\{A, B, C, D\}$$

$$E(G)=\{\{A, B\}, \{A, D\}, \{B, C\}, \{B, D\}\}$$

**Undirected Graph**  $G=\{\{A, B, C, D\}, \{\{A,B\}, \{A,D\}, \{B,C\}, \{B, D\}\}\}$

## Directed Graph

- When the edges in a graph have a direction, the graph is called *directed graph* (or *digraph*)  $G(V, E)$  consists of:
  - A nonempty set of vertices (V)
  - A set of *directed edges* (arcs) (E)
- Each directed edge is associated with an ordered pair of vertices. For example, the ordered pair (u, v) is said to *start* at u and *end* at v. **Note:  $(1, 2) \neq (2, 1)$**



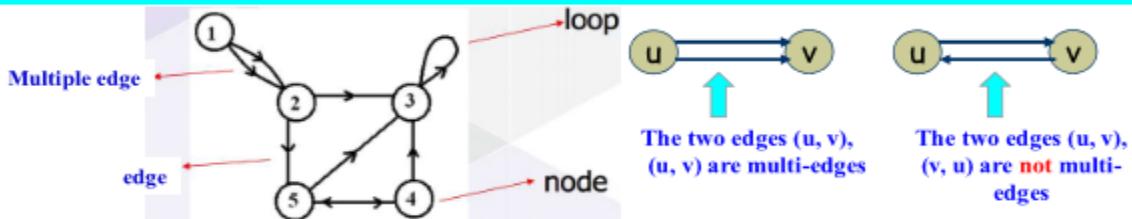
$$V(G)=\{0, 1, 2, 3\}$$

$$E(G)=\{(1,0), (1,2), (1,3), (0,2), (0,3), (2,0), (2,1), (3, 2), (3, 1)\}$$

**Directed Graph**

$G=\{\{0,1,2,3\}, \{(1,0), (1,2), (1,3), (0,2), (0,3), (2,0), (2,1), (3, 2), (3, 1)\}\}$

# Graph Theory Terminology



- The line drawn between two adjacent nodes is called an **edge**.
- Suppose edge  $e = [u, v]$ , then the nodes  $u$  and  $v$  are called **end points** of the edge  $e$ .
- The node  $u$  is called **source** node and node  $v$  is called **destination** node,
- **Multiple edges:** Distinct edges  $e$  and  $e'$  are called multiple edges if they connect the same endpoints, that is, if  $e = [u, v]$  and  $e' = [u, v]$ .
- **Loops:** An edge  $e$  is called a loop if it has identical endpoints, that is, if  $e = [u, u]$ .

11

## Types of Undirected Graphs

12

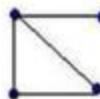
120

## Types of undirected Graphs

### Simple Graphs

Each edge connects **two different** vertices

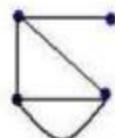
No two edges connect the same pair of vertices



simple graph

### Multigraphs

Graphs that may have multiple edges connecting the same vertices

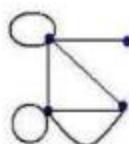


multigraph

### Pseudographs

Graphs that may include loops

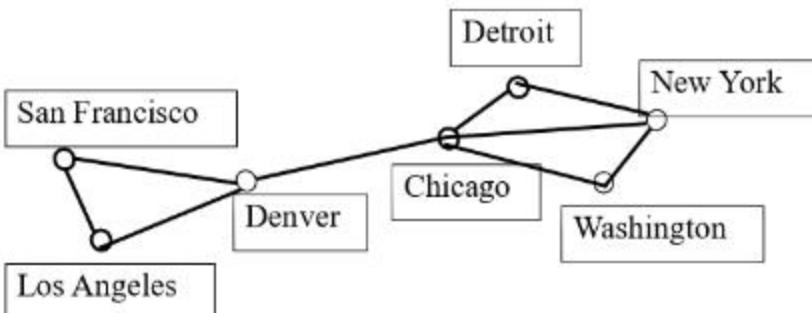
Possibly multiple edges connecting the same pair of vertices



pseudograph

13

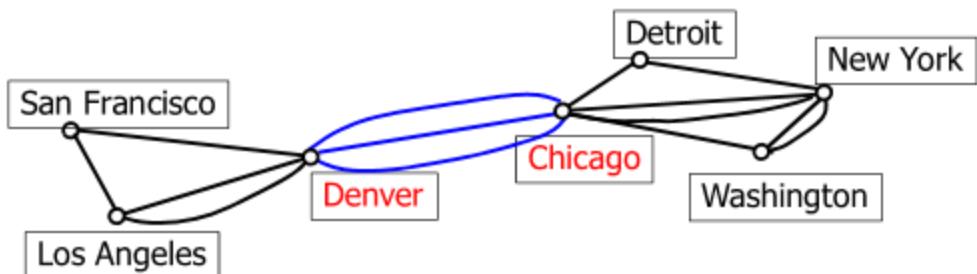
## Simple Graph Example



- This simple graph represents a network that contains computers and telephone links between computers.

14

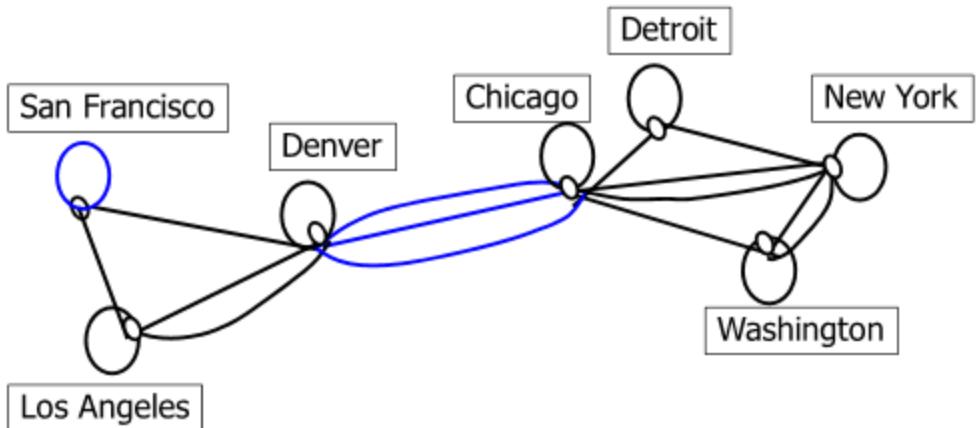
## Multi-graph Example



- There can be multiple telephone lines between two computers in the network.

15

## Pseudo-graph Example



- There can be telephone lines in the network from a computer to itself.

16

## Directed Graph Types

17

### Types of directed Graphs

#### Simple Directed Graph

#### Directed Multigraph

No loops

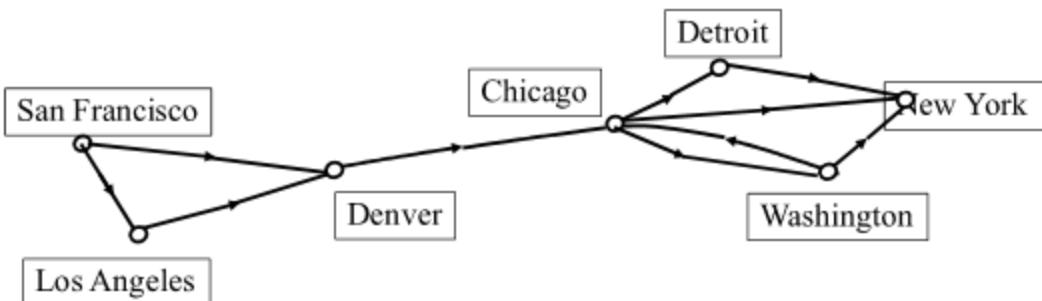
multiple edges in  
the same direction  
are not allowed.

Directed graphs  
that may have  
**multiple directed  
edges from a vertex  
to a second  
(possibly the same)**

18

## Simple Directed Graph Example

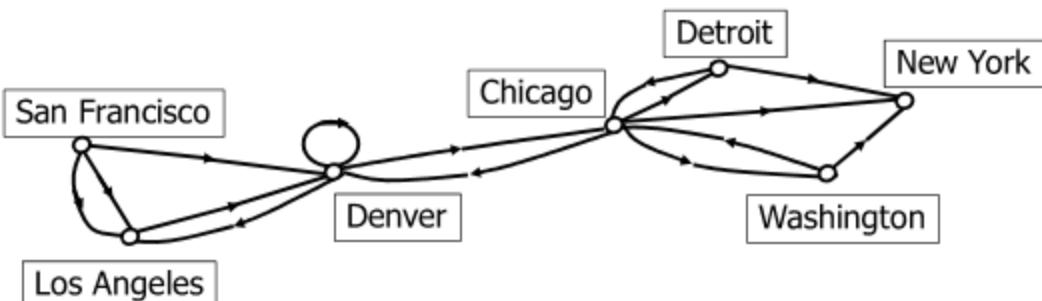
- The edges are ordered pairs of (not necessarily distinct) vertices.



Some telephone lines in the network may operate in only one direction. Those that operate in two directions are represented by pairs of edges in opposite directions.

19

## Directed Multigraph Example

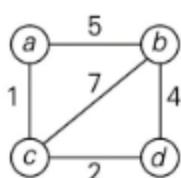


There may be several one-way lines in the same direction from one computer to another in the network.

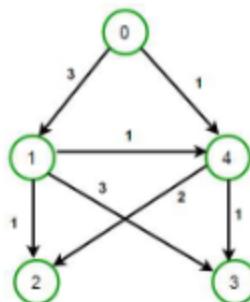
20

## Weighted and Labelled Graph

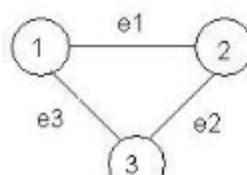
- **A Weighted Graph:** is a graph or digraphs G with **numbers** assigned to its **edges**  $W(e)$ . These numbers are called **weights** or **costs** of the edge.
  - Used to find shortest path between two points.
  - If an edge does not have any weight then the weight is considered as 1.
- **Labelled Graph :** A graph or digraphs G is said to be labelled if its edges are assigned data.



(a) Weighted graph



(b) Weighted digraph



(c) Labelled graph

21

## Graph Model Structure

Three key questions can help us understand the structure of a graph:

1. Are the edges of the graph undirected or directed (or both)?
2. If the graph is undirected, are the multiple edges present that connect the same pair of vertices?
3. If the graph is directed, are multiple directed edges present?
4. Are the loops present?

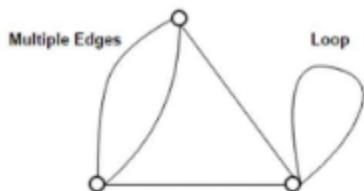
# Summary

Type	Edges	Multiple Edges Allowed?	Loops Allowed?
Simple Graph	Undirected	NO	NO
Multigraph	Undirected	YES	NO
Pseudograph	Undirected	YES	YES
Simple Directed Graph	Directed	NO	NO
Directed Multigraph	Directed	YES	YES

23

## Example (1)

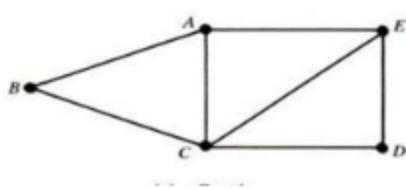
What is the type of each Graph?



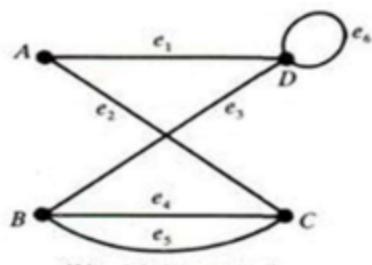
(a) .....



(b) .....



(c) .....

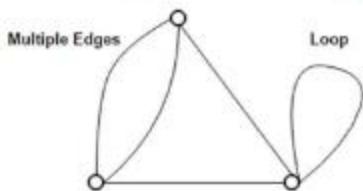


(d) .....

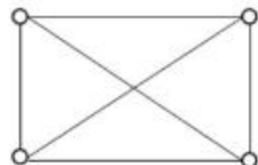
24

## Example (1)

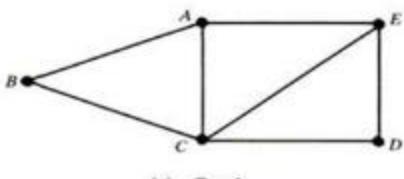
What is the type of each Graph?



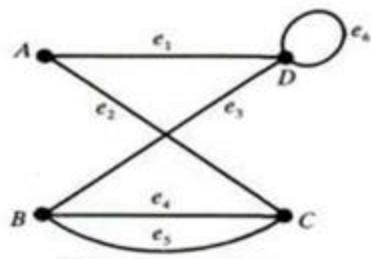
(a) Pseudograph



(b) Simple Graph



(c) Simple Graph

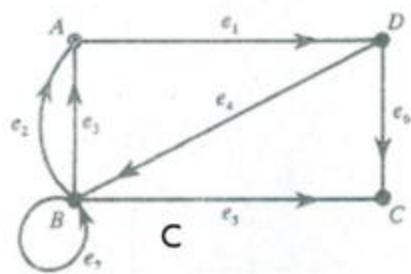
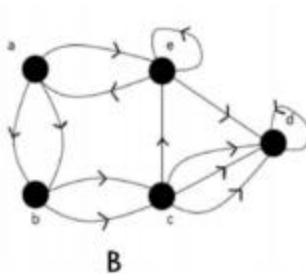
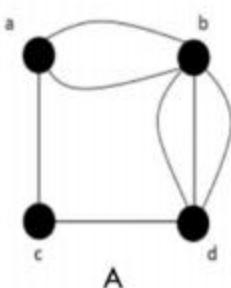


(d) Labelled Pseudograph

25

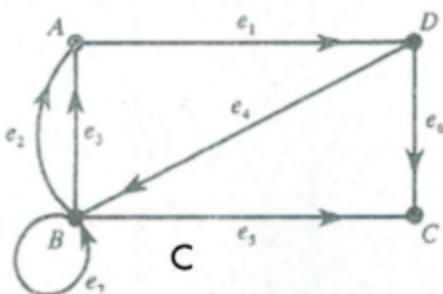
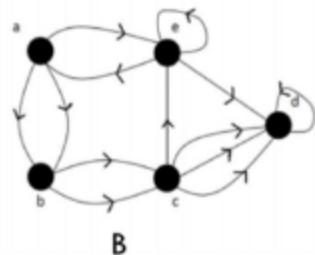
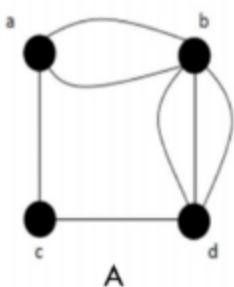
## Example (2)

- Determine whether the graphs shown below have
  - directed or undirected edges,
  - multiple edges,
  - one or more loops



26

## Example (2): Solution



Graph	directed or undirected	multiple edges	Loops	Type
A	undirected	multi-graph	no loops	Multi-graph
B	directed	directed multi-graph	two loops	Directed Multi-graph
C	Labeled directed	Labeled directed multi-graph	one loop	Labeled Directed Multi-graph

27

## Graph Terminology

- Adjacent vertex
- Incident edge from vertex
- Degree of a vertex
- Isolated, pendant, odd, even vertex
- In-degree and Out-degree of a vertex

28

## Undirected Graphs

29

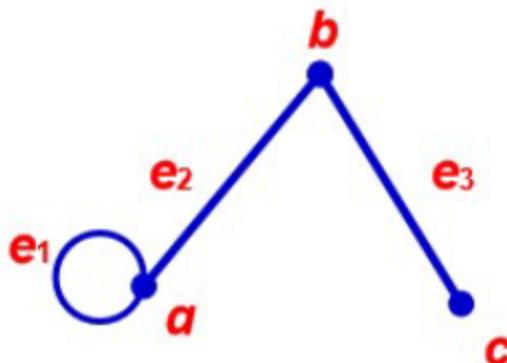
### Adjacent (متجاورة) Vertices in undirected Graphs

- Two vertices,  $u$  and  $v$  in an undirected graph  $G$  are called *adjacent* (or *neighbors*) in  $G$ , if  $\{u, v\}$  is an edge of  $G$ .
- An edge  $e$  *connecting*  $u$  and  $v$  is called *incident* تابعة/واقعة/متواجدة with vertices  $u$  and  $v$ , or is said to connect  $u$  and  $v$ .
- The vertices  $u$  and  $v$  are called *endpoints* of edge  $\{u, v\}$ .

30

### Example (3)

- a and b are adjacent.
- a and c not adjacent.
- $e_2$  is incident with a and b .



31

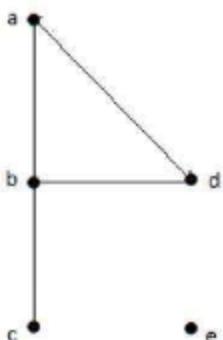
### Degree of a Vertex

- The *degree of a vertex* in an undirected graph is the number of edges *incident* المتواجدة with it, **except** that a **loop** at a vertex contributes **twice** to the degree of that vertex
- The **degree of a vertex**  $v$  is denoted by **deg(v)** is the number of edges containing  $v$ .

32

## Isolated (معزول) Vertex : $\deg(u) = 0$

- A vertex of degree zero ( $\deg(u) = 0$ ) is called **isolated**.
- It follows that an isolated vertex is not **adjacent** to any vertex.
- If  $\deg(u) = 0$  — that is, if  $u$  does not belong to any edge—then  $u$  is called an **isolated node**.

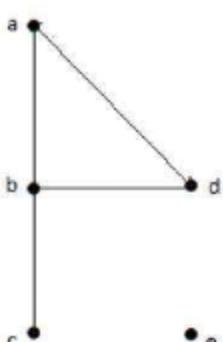


Vertex e is an isolated vertex :  $\deg(e) = 0$

33

## Pendant (معلق) Vertex : $\deg(u) = 1$

- A vertex is **pendant** معلق if and only if it has degree one.
- Consequently, a pendant vertex is adjacent to exactly one other vertex.

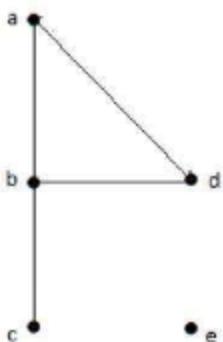


Vertex c is pendent vertex :  $\deg(c) = 1$

34

## Even and odd Vertex

- A vertex with an even number of edges attached to it is an even vertex.
- A vertex with an odd number of edges attached to it is an odd vertex.



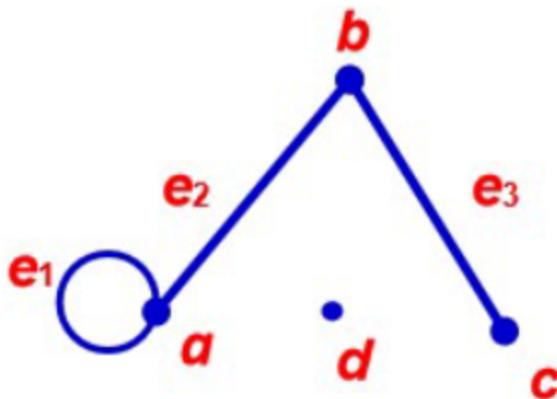
Vertex a is an even vertex :  $\deg(a) = 2$

Vertex b is an odd vertex :  $\deg(b) = 3$

35

## Example (4)

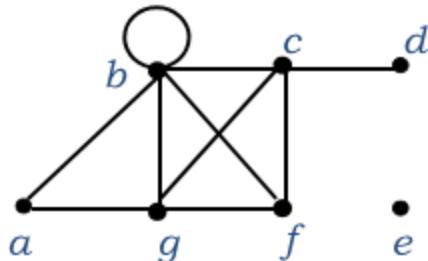
- $\deg(a) = 3$  it is an odd vertex
- $\deg(b) = 2$  an even vertex
- $\deg(c) = 1$ , so vertex c is pendant
- $\deg(d) = 0$ , so vertex d is isolated



36

## Example (5)

- Find the degrees of all the vertices.
- Identify the type of each vertex (isolated, pendant, even, odd)



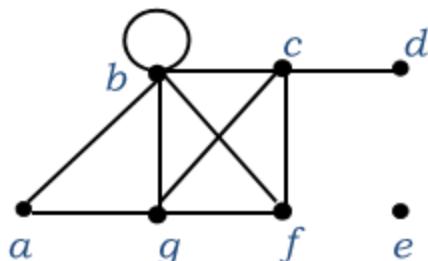
Solution:

Node (v)	deg(v)	Type
a	.....	.....
b	.....	.....
c	.....	.....
d	.....	.....
e	.....	.....
f	.....	.....
g	.....	.....

37

## Example (5)

- Find the degrees of all the vertices.
- Identify the type of each vertex (isolated, pendant, even, odd)



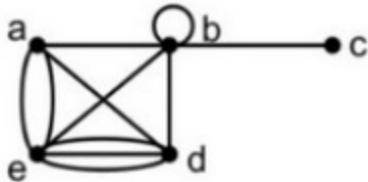
Solution:

Node (v)	deg(v)	Type
a	2	even
b	6	even
c	4	even
d	1	pendant
e	0	isolated
f	3	odd
g	4	even

38

## Exercise (1)

- What are the degrees of the vertices in this graph
- Identify the type of each vertex (isolated, pendant, even, odd)



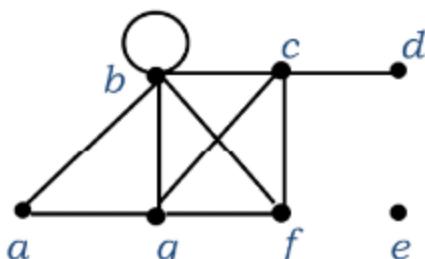
Solution:

Node (v)	deg(v)	type
a	.....	.....
b	.....	.....
c	.....	.....
d	.....	.....
e	.....	.....

39

## The Handshaking Theorem

- Let  $G=(V, E)$  be an undirected graph with  $E$  edges.  
Then  $2|E| = \sum_{v \in V} \deg(v)$



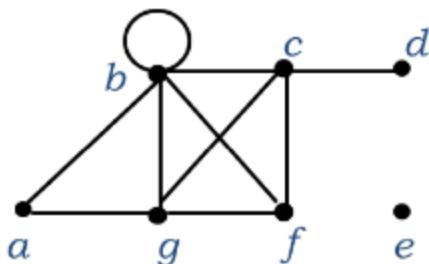
$$|E|=10$$

$$\sum_{v \in V} \deg(v) = 20$$

Node (v)	deg(v)
a	2
b	6
c	4
d	1
e	0
f	3
g	4
	20

40

## Example (5)



Node (v)	deg(v)
a	2
b	6
c	4
d	1
e	0
f	3
g	4

The number of edges in the above graph is:  $|E|=10$

And the sum of degrees for all vertices is:  $\sum_{v \in V} \deg(v) = 20$

Therefore, the handshaking theorem is satisfied

$$2|E| = \sum_{v \in V} \deg(v)$$

41

## Example (6)

How many edges are there in a graph with 10 vertices, each of degree 6?

□ Solution:

According the Handshaking Theorem

$$2|E| = 6 \cdot 10 = 60 \Rightarrow |E| = 30$$

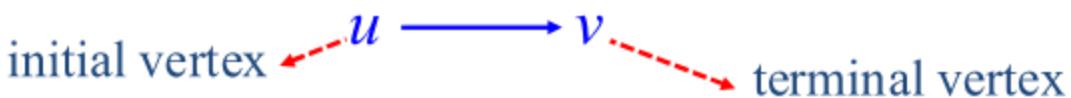
42

## Directed Graphs

43

### Adjacent Vertices in Directed Graphs

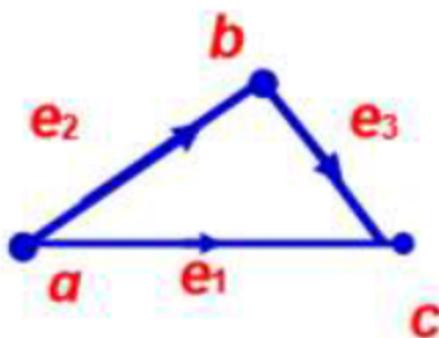
- When  $(u, v)$  is an edge of a directed graph  $G$ ,
  - $u$  is said to be **adjacent to** (Neighbor)  $v$  .
  - $v$  is said to be **adjacent from**  $u$  .



44

## Example (7)

- a is adjacent to b
- b is adjacent to c
- c is adjacent from a (b)
- Vertex a is initial vertex of (a, b)
- Vertex b is terminal vertex of (a, b)



45

## Degree of a Vertex

### □ In-degree of a vertex v

- The number edges **pointing towards** the node v are called in-degree of v.
- The number of vertices adjacent to v (The number of edges **ending** at v)
- Denoted by **deg<sup>-</sup>(v)**

### □ Out-degree of a vertex v

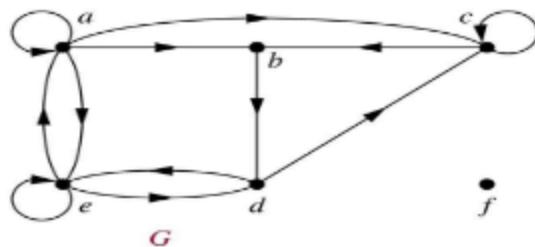
- The number edges **pointing away** from the node are called out-degree/out-order.
- The number of vertices adjacent from v (the number of edges with v as their **initial** vertex: beginning at v)
- Denoted by **deg<sup>+</sup>(v)**

### □ **A loop at a vertex contributes 1 to both the in-degree and out-degree.**

46

## Example (8)

Find the in-degrees and out-degrees of this digraph.

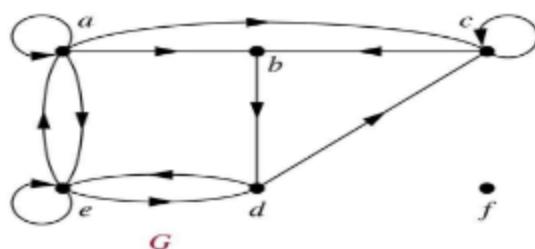


Node	In-degrees (deg <sup>-</sup> )	Out-degrees (deg <sup>+</sup> )
a	.....	.....
b	.....	.....
c	.....	.....
d	.....	.....
e	.....	.....
f	.....	.....

47

## Degree of a Vertex : Example (8)

Find the in-degrees and out-degrees of this digraph.



Node	In-degrees (deg <sup>-</sup> )	Out-degrees (deg <sup>+</sup> )
a	2	4
b	2	1
c	3	2
d	2	2
e	3	3
f	0	0

48

## Theorem

- Let  $G = (V, E)$  be a graph with directed edges.

Then:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$$

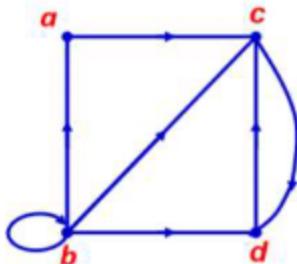
The sum of the in-degrees of all vertices in a digraph  
= the sum of the out-degrees  
= the number of edges. (an edge is in for a vertex and  
out for the other)

49

## Example (9)

For following directed multi-graph, determine:

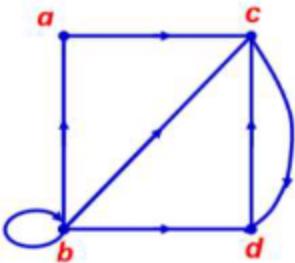
1. The number of vertices and edges.
2. The in-degree and out-degree of each vertex.
3. The sum of the in-degrees of the vertices and
4. The sum of the out-degrees of the vertices directly.
5. Show that they are both equal to the number of edges in graph.



Prove that  $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$

50

## Example (9)



1. The in-degrees in G are:

- $\deg^-(a) = 1$ ,  $\deg^-(b) = 1$
- $\deg^-(c) = 3$ ,  $\deg^-(d) = 2$
- $\sum_{v \in V} \deg^-(v) = 7$

2. The out-degrees in G are:

- $\deg^+(a) = 1$ ,  $\deg^+(b) = 4$
- $\deg^+(c) = 1$ ,  $\deg^+(d) = 1$
- $\sum_{v \in V} \deg^+(v) = 7$

3. The number of edges

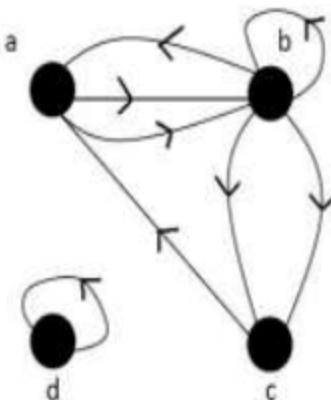
$$|E| = 7$$

51

## Example (10)

□ For following directed multi-graph, determine:

1. The number of **vertices** and **edges**.
2. The **in-degree** and **out-degree** of each vertex.
3. The **sum** of the **in-degrees** of the vertices and
4. The **sum** of the **out-degrees** of the vertices directly.
5. Show that they are both equal to **the number of edges** in graph.



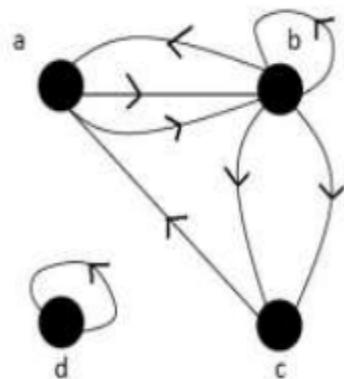
52

## Example (10): Solution

1.  $|V|= 4, |E|= 8.$
2. The degrees are :

$$\begin{array}{ll} \deg^-(a) = 2, & \deg^+(a) = 2, \\ \deg^-(b) = 3, & \deg^+(b) = 4, \\ \deg^-(c) = 2, & \deg^+(c) = 1, \\ \deg^-(d) = 1, & \deg^+(d) = 1. \end{array}$$

3. The sum of the in-degrees of the vertices = 8
4. The sum of the out-degrees of the vertices = 8,
5.  $|E|=8=$  The sum of the in/out-degrees of the vertices = 8



53

## Degree of a graph

### Definition:

The degree of a graph is the sequence of the degrees of the vertices of the graph in non-increasing order.

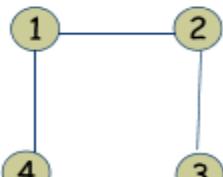
54

## Some Special Classes of a Simple Graph

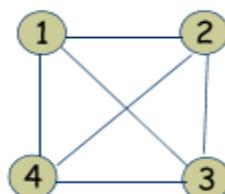
7

### Regular Graph

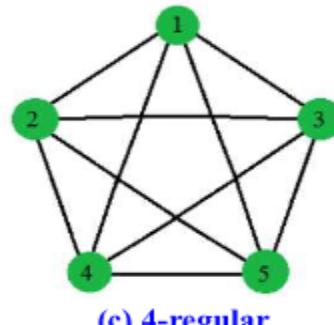
- A simple graph  $G=(V, E)$  is called **regular graph** if every vertex has the same degree.
- A regular graph is called **n-regular** if  $\deg(v)=n, \forall v \in V$ .
- Here is some examples of regular graphs of degree 2, 3 and 4, respectively.



(a) 2-regular



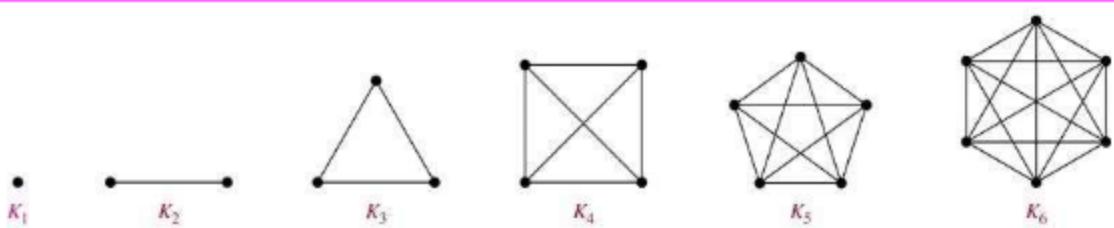
(b) 3-regular



(c) 4-regular

8

## Complete graph



- The **complete graph** on  $n$  vertices ( $K_n$ ) is the **simple graph** that contains exactly one edge between **each pair** of **distinct** vertices.
- **Complete Graph :** A graph is called complete if all the nodes of the graph are adjacent to each other.
- The figures above represent the complete graphs,  $K_n$ , for  $n = 1, 2, 3, 4, 5$ , and 6.

9

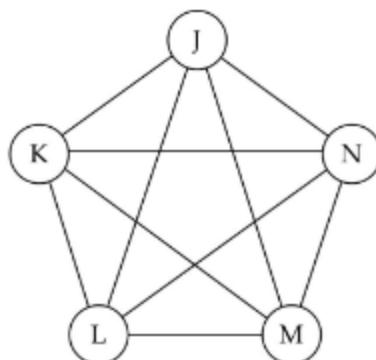
## Example (12)

Draw a complete undirected graph ( $G_1$ ) with 5 vertices {K, L, M, N, J}, what is the number of edges in  $G_1$ ?

The answer

The number of edges in a complete undirected graph with  $n$  vertices=  **$n*(n-1)/2$  edges**.

The number of edges=  $6*5/2=15$



Complete undirected graph

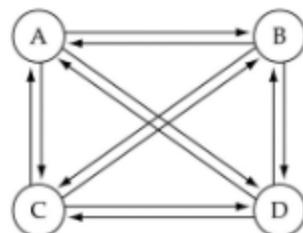
## Example (12)

Draw a complete directed graph ( $G_2$ ) with 4 vertices {A, B, C, D}, what is the number of edges in  $G_2$ ?

### The answer

The number of edges in a complete directed graph with n vertices =  **$n*(n-1)$  edges.**

The number of edges in a graph with 4 vertices =  $4*3=12$

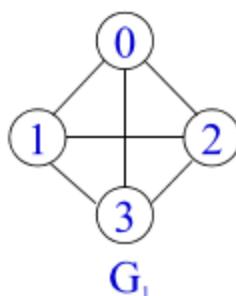


Complete directed graph

## Example (13-1)

Draw a complete undirected graph ( $G_1$ ) with 4 vertices {0, 1, 2, 3}, what is the number of edges in  $G_1$ ?

### The answer



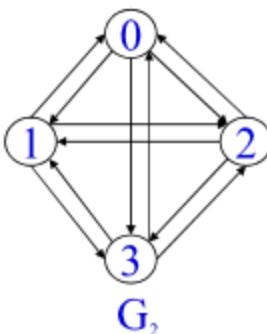
complete undirected graph

complete undirected graph:  $n(n-1)/2$  edges =  $4*3/2=6$

## Example (13-2)

Draw a complete directed graph ( $G_2$ ) with 4 vertices  $\{0, 1, 2, 3\}$ , what is the number of edges in  $G_2$ ?

The answer



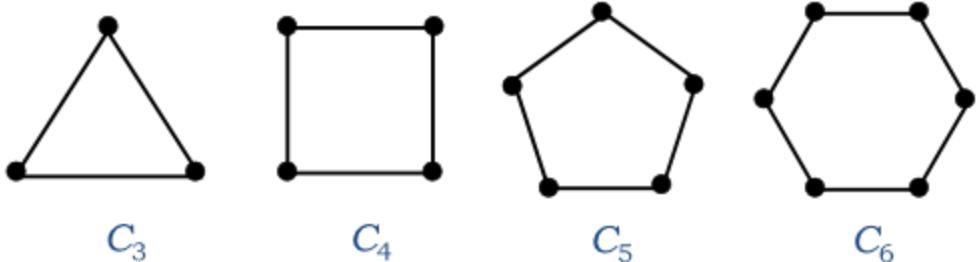
complete directed graph

complete directed graph:  $n(n-1)$  edges       $=4*3=12$

## CYCLE

- The *cycle*  $C_n$  ( $n \geq 3$ ), consists of  $n$  vertices  $v_1, v_2, \dots, v_n$  and edges  $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}$ , and  $\{v_n, v_1\}$ .

Cycles:

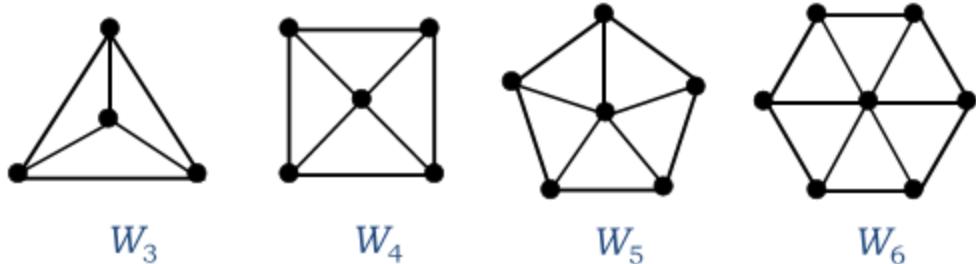


- Cycle: A cycle is closed path with length 3 or more. A cycle of length  $k$  is called a  $k$ -cycle.

## WHEEL

- When a new vertex is added to a cycle  $C_n$  and this new vertex is connected to each of the  $n$  vertices in  $C_n$ , we obtain a *wheel*  $W_n$ .

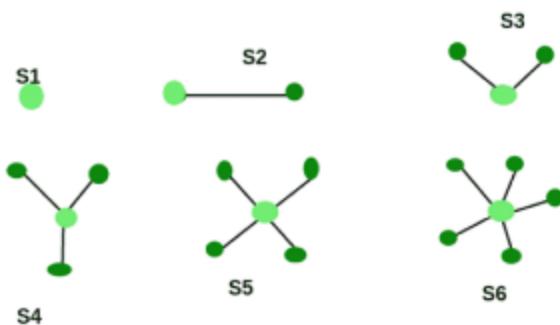
Wheels:



15

## Star Graph

- Star graph :** Star graph is a special type of graph in which :
  - $n-1$  vertices have degree 1 and
  - a single vertex have degree  $n - 1$ .
- This looks like that  $n - 1$  vertices are connected to a single central vertex.
- A star graph with total  $n$  vertex is written as  $S_n$ .

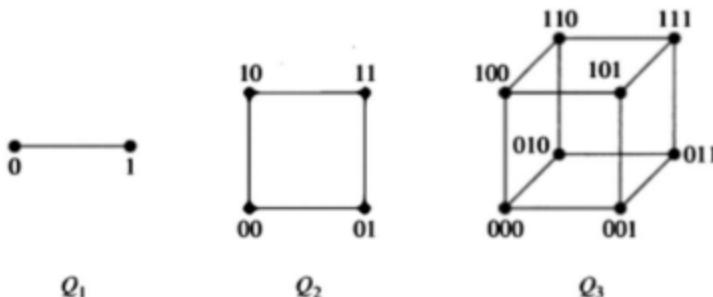


Star graph of order- $n$  ( $S_{n+1}$ )

16

## n-Cubes The n-dimensional hypercube

- n-cube, denoted by  $Q_n$ , is the graph that has vertices representing the  $2^n$  bit strings of length n.
- Two vertices are adjacent if and only if the bit strings that they represent differ in exactly one bit position.
- The graphs  $Q_1$ ,  $Q_2$ , and  $Q_3$  are displayed in this figure.

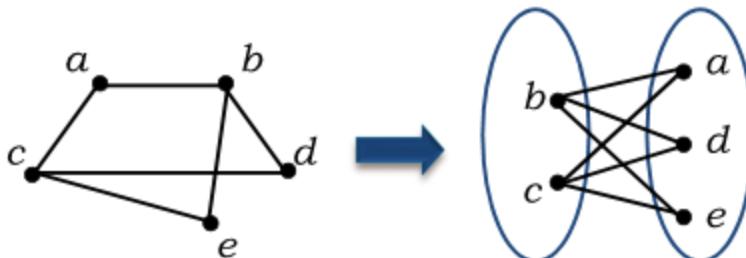


The n-cube  $Q_n$  for  $n = 1, 2$ , and 3.

17

## Bipartite Graph

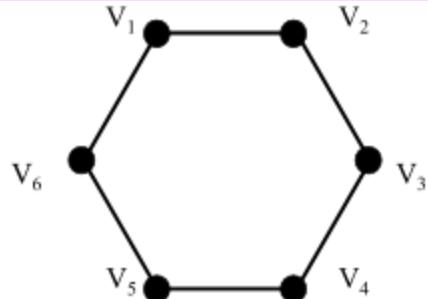
- A simple graph is called **bipartite** if its vertex set  $V$  can be partitioned into two disjoint nonempty sets  $V_1$  and  $V_2$ ,  $V_1 \cap V_2 = \emptyset$  such that every edge in the graph connects a vertex in  $V_1$  and a vertex in  $V_2$  (so that no edge in  $G$  connects either two vertices in  $V_1$  or two vertices in  $V_2$ ).



18

## Example (14)

Is  $C_6$  Bipartite?

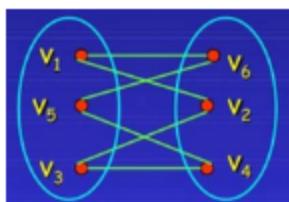


The answer

Yes. Why?

Because:

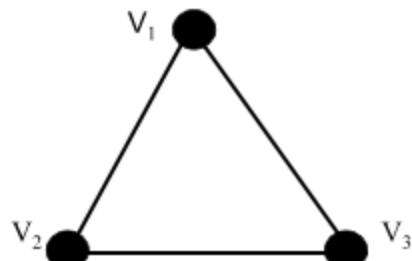
- ✓ its vertex set can be partitioned into the two sets  
 $V_1 = \{v_1, v_3, v_5\}$  and  $V_2 = \{v_2, v_4, v_6\}$
- ✓ every edge of  $C_6$  connects a vertex in  $V_1$  with a vertex in  $V_2$



19

## Example (15)

Is  $K_3$  Bipartite?



The answer:

No. Why not?

Because:

- ✓ There is no way to partition the vertices into two sets so that there are **no edges** with both endpoints in the same set
- ✓ Each vertex in  $K_3$  is connected to every other vertex by an edge
- ✓ If we divide the vertex set of  $K_3$  into two disjoint sets, one set must contain two vertices
- ✓ These two vertices are connected by an edge
- ✓ **Therefore, this graph is not bipartite**

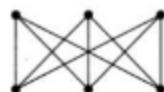
20

## Complete Bipartite Graphs

- The complete bipartite graph  $K_{m,n}$  is the graph that has its vertex set partitioned into two subsets of  $m$  and  $n$  vertices, respectively.
- There is an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset.



$K_{2,3}$



$K_{3,3}$



$K_{3,5}$



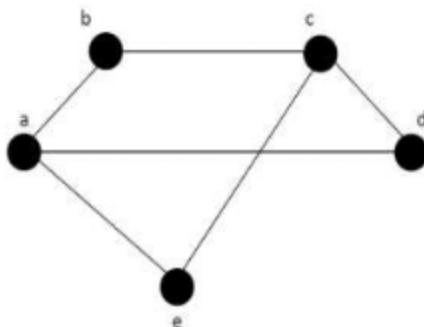
$K_{2,6}$

The complete bipartite graphs  $K_{2,3}$ ,  $K_{3,3}$ ,  $K_{3,5}$ , and  $K_{2,6}$

21

## Example (16)

Determine whether the graph shown below is bipartite.  
Justify your answer.



**The answer:**

The graph is bipartite, its vertex set can be partitioned into two sets :

$$V_1 = \{ a, c \} \text{ and } V_2 = \{ b, d, e \}.$$

22

## Graph Operations

Subgraph  
Union  
Complement  
Intersection

23

### Subgraph

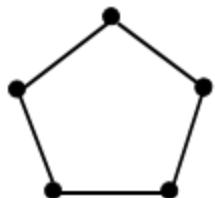
- A **subgraph** of a graph  $G = (V, E)$  is a graph  $G'=(V',E')$  where  $V' \subseteq V$  and  $E' \subseteq E$  and we write  $G' \subseteq G$ .

24

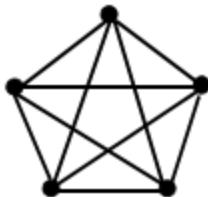
## Example (17)

Is  $C_5$  a subgraph of  $K_5$ ?

The answer:



$C_5$



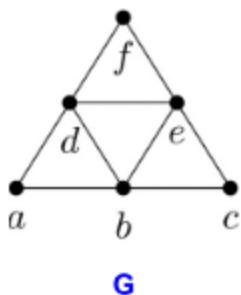
$K_5$

Let  $C_5$  represented by  $G' = (V', E')$ ,  
Since  $V' = V$  and  $E' \subseteq E \rightarrow C_5 \subseteq K_5$

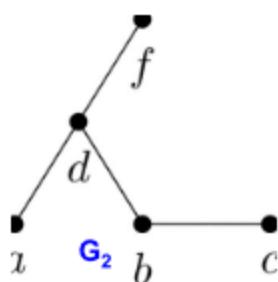
25

## Example (18)

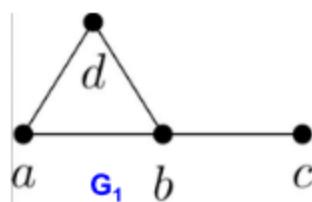
Consider the graph  $G$  is as follow



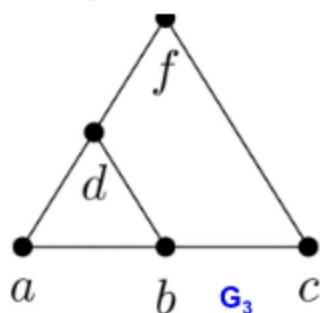
2. Is  $G_2$  a subgraph of  $G$ ?



1. Is  $G_1$  a subgraph of  $G$ ?



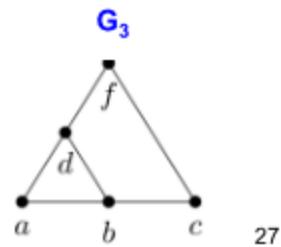
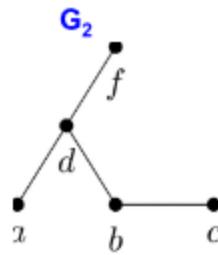
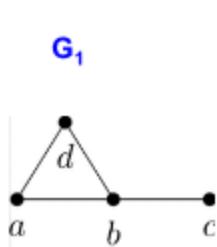
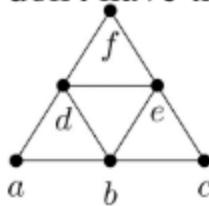
3. Is  $G_3$  a subgraph of  $G$ ?



26

## Example (18): Answer

- Here both  $G_1$  and  $G_2$  are subgraphs of  $G$ .
- The graph  $G_3$  is NOT a subgraph of  $G$ , even though it looks like all we did is remove vertex  $e$ .
- The reason is that in  $E_3$  we have the edge  $\{c, f\}$  but this is not an element of  $E$ , so we don't have the required  $E_3 \subseteq E_1$ .



27

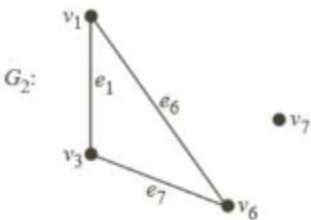
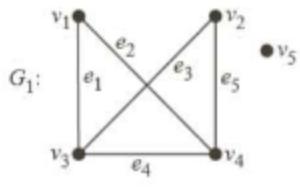
## Union

- The *union* of 2 simple graphs  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  is the simple graph with vertex set  $V = V_1 \cup V_2$  and edge set  $E = E_1 \cup E_2$ .
- The union is denoted by  $G_1 \cup G_2$ .

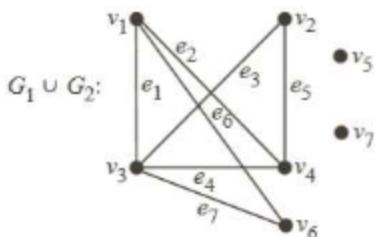
28

## Example (19)

find  $G_1 \cup G_2$



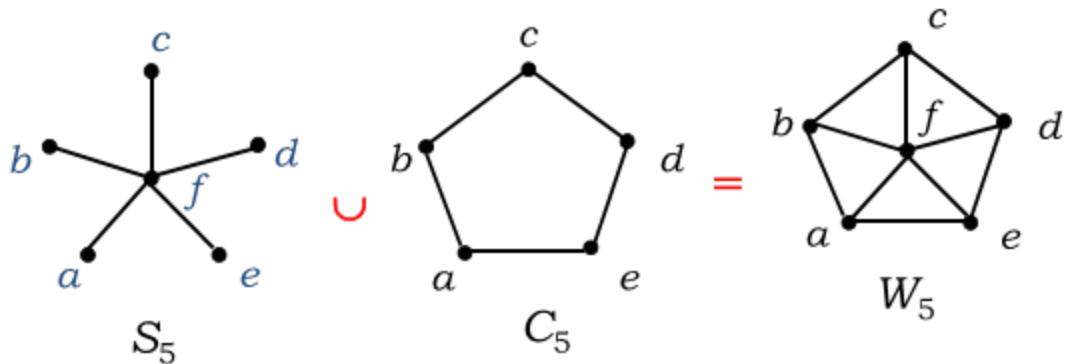
The answer



29

## Example (20)

□ Find the union of the graph  $S_5$  and the graph  $C_5$



The answer

$$S_5 \cup C_5 = W_5$$

30

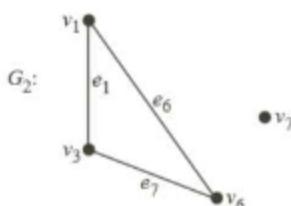
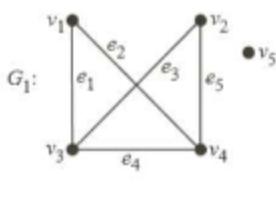
## Intersection

- The **intersection** of 2 simple graphs  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  is the simple graph with vertex set  $V_1 \cap V_2$  and edge set  $E_1 \cap E_2$ .
- The **intersection** is denoted by:  
$$G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$$

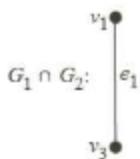
31

## Example (21)

find  $G_1 \cap G_2$



The answer



32

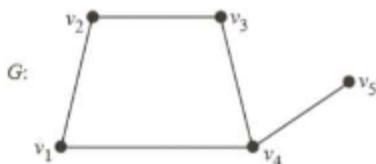
## Complement

The **complement** of the simple graph  $G = (V, E)$  is the simple graph  $\bar{G} = (V, \bar{E})$ , where the edges in  $\bar{E}$  are exactly the edges not in  $G$ .

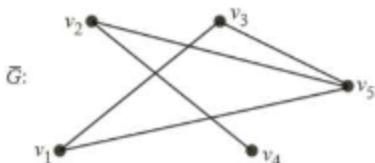
33

### Example (22)

Find the complement  $\bar{G}$  of the graph  $G$



**The answer**



34

## Example (23)

Find the complement of the graph  $K_n$

**The answer**

The **complement** of the complete graph  $K_n$  is the empty graph with  $n$  vertices.

35

## Removal of a vertex

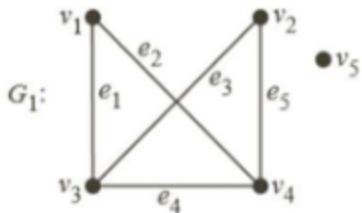
**Definition:**

If  $v$  is a vertex of the graph  $G = (V, E)$ , then  $G - v$  is the subgraph of  $G$  induced by the vertex set  $V - \{v\}$ . We call this operation the removal of a vertex.

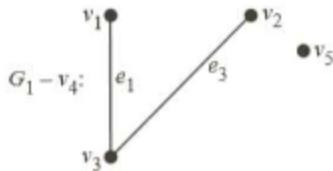
36

## Example (24)

Draw all subgraphs  $G = G_1 - v_4$  of the graph  $G_1$



The answer



37

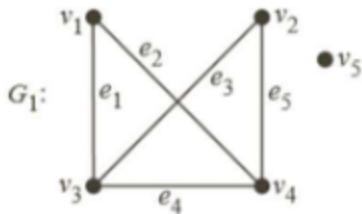
## Removal of an edge

Similarly, if  $e$  is an edge of the graph  $G = (V, E)$ , then  $G - e$  is graph  $(V, E')$ , where  $E'$  is obtained by removing  $e$  from  $E$ . This operation is known as removal of an edge.

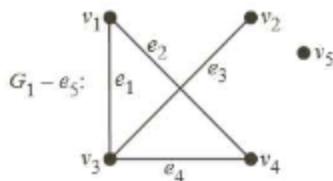
38

## Example (25)

Draw all subgraphs  $G=G_1 - e_5$  of the graph  $G_1$



The answer



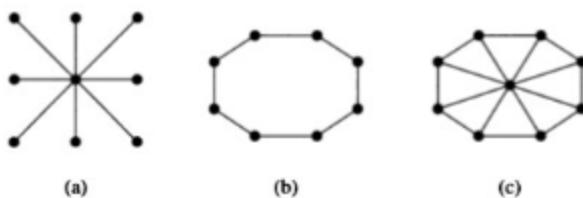
39

## Some Applications of Special Types of Graphs

### Local Area Networks (LAN)

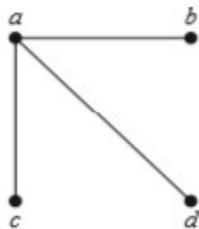
- LANs are based on a **star** topology, where all devices are connected to a central control device. A LAN can be represented using a complete bipartite graph  $K_{l,n}$ , as shown in figure (a) (Note that  $S_{n+1} = K_{l,n}$ )
- LANs are based on a **ring** topology, where each device is connected to exactly two others. LANs with a **ring** topology are modeled using  $n$ -cycles,  $C_n$ , as in figure (b).
- LANs can be a hybrid of these two topologies. Messages may be sent around the ring, or through a central device. This **redundancy** makes the network more **reliable**. This LANs can be modeled using wheels  $W_n$ , as in figure (c).

$$S_9 \cup C_8 = W_8$$

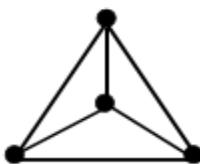


## Exercise (1): check your understanding

Draw all subgraphs of this graph



How many subgraphs with at least one vertex does  $W_3$  have?

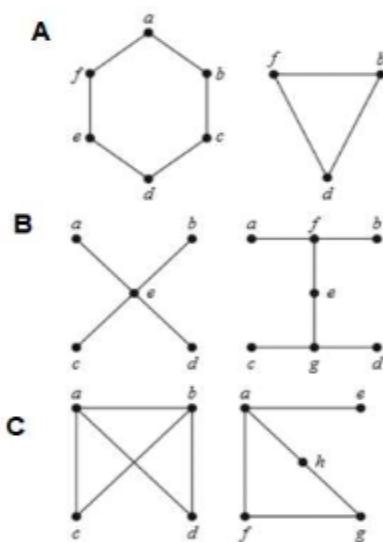


$W_3$

41

## Exercise (2) : check your understanding

In Figures A-C find the union of the given pair of simple graphs.(Assume edges with the same endpoints are the same.)



42

## Paths and cycles

### Paths

- Informally, a path is a **sequence of edges** that begins at a **vertex** of a graph and travels from vertex to vertex along **edges** of the graph.
- As the path travels along its edges, it visits the **vertices** along this path, that is, the endpoints of these edges.

## Definition of path and cycle for undirected graph

A formal definition of paths and related terminology is given as:

Let  $n$  be a nonnegative integer and  $G$  an undirected graph.

A **path of length  $n$**  from  $u$  to  $v$  in  $G$  is a sequence of  $n$  edges  $e_1, \dots, e_n$  of  $G$  for which there exists a sequence  $x_0 = u, x_1, \dots, x_{n-1}, x_n = v$  of vertices such that  $e_i$  has, for  $i = 1, \dots, n$ , the endpoints  $x_{i-1}$  and  $x_i$ .

## Definition of path and cycle for undirected graph

When the graph is simple, we denote this path by its vertex sequence  $x_0, x_1, \dots, x_n$  (because listing these vertices uniquely determines the path).

## Definition of path and cycle for undirected graph

The path is a **cycle** if it begins and ends at the same vertex, that is, if  $u = v$ , and has length greater than zero.

The path or **cycle** is said to *pass through* the vertices  $x_1, x_2, \dots, x_{n-1}$  or *traverse* the edges  $e_1, e_2, \dots, e_n$ .

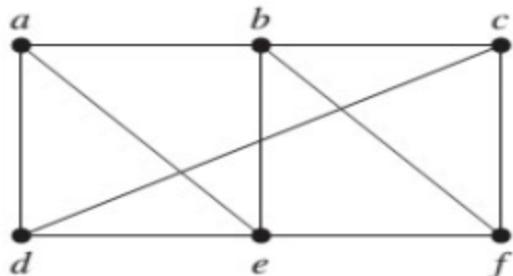
**Note that:**

A path or **cycle** is **simple** if it does not contain the same edge more than once.

## Example (26)

Determine whether each of these is **path** / **cycle** / **not a path**.

- (1)  $a, d, c, f, e$
- (2)  $d, e, c, a$
- (3)  $b, c, f, e, b$
- (4)  $a, b, e, d, a, b$



### The answer

- (1)  $a, d, c, f, e$  is a **simple path** of length 4, because  $\{a, d\}$ ,  $\{d, c\}$ ,  $\{c, f\}$ , and  $\{f, e\}$  are all edges.
- (2)  $d, e, c, a$  is **not a path**, because  $\{e, c\}$  is **not an edge**.
- (3)  $b, c, f, e, b$  is a **cycle** of length 4 because  $\{b, c\}$ ,  $\{c, f\}$ ,  $\{f, e\}$ , and  $\{e, b\}$  are edges, and this path begins and ends at b.
- (4) The path  $a, b, e, d, a, b$ , which is of length 5, is **not simple** because it contains the edge  $\{a, b\}$  twice.

## Definition of path and cycle for directed graph.

Let  $n$  be a nonnegative integer and  $G$  a directed graph.

- A *path* of length  $n$  from  $u$  to  $v$  in  $G$  is a sequence of edges  $e_1, e_2, \dots, e_n$  of  $G$  such that  $e_1$  is associated with  $(x_0, x_1)$ ,  $e_2$  is associated with  $(x_1, x_2)$ , and so on, with  $e_n$  associated with  $(x_{n-1}, x_n)$ , where  $x_0 = u$  and  $x_n = v$ .
- When there are no multiple edges in the directed graph, this path is denoted by its vertex sequence  $x_0, x_1, x_2, \dots, x_n$ .

## Definition of path and cycle for directed graph.

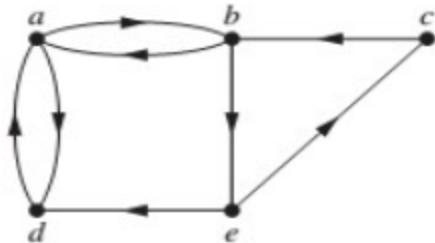
- A path of length greater than zero that begins and ends at the same vertex is called a *cycle*.
- A path or *cycle* is called *simple* if it does not contain the same edge more than once.

## Example (27)

Does each of these lists of vertices form a path in the following graph?

Which paths are simple? Which are **cycles**? What are the lengths of those that are paths?

- a)  $a, b, e, c, b$
- b)  $a, d, a, d, a$
- c)  $a, d, b, e, a$
- d)  $a, b, e, c, b, d, a$



**Connectedness in Undirected Graphs**

## Connectedness in Undirected Graphs

- An undirected graph is called **connected** if there is a path between every pair of distinct vertices of the graph.
- An undirected graph that is **not connected** is called **disconnected**.
- We say that we **disconnect** a graph when we **remove** vertices or edges, or both, to produce a **disconnected** subgraph.

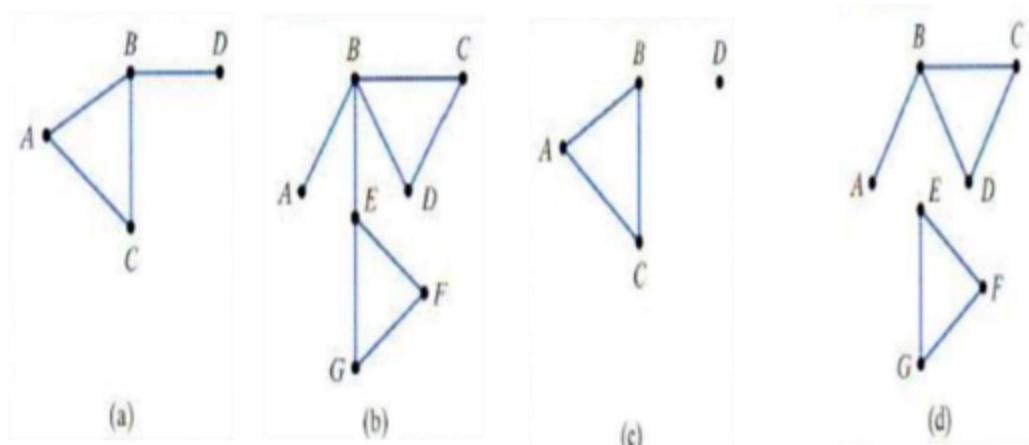
Thus, any two computers in the network can communicate if and only if the graph of this network is connected.

## Connected and Disconnected Graph

- The words **connected** and **disconnected** are used to describe graphs.
- A graph is **connected** if for any two of its vertices there is **at least one path connecting** them.
- Thus, a graph is connected if it **consists of one piece**.
- If a graph is **not connected** it is said to be **disconnected**.
- A disconnected graph is made up of **pieces that are by themselves connected**.
- Such pieces are called the **components** of the graph.

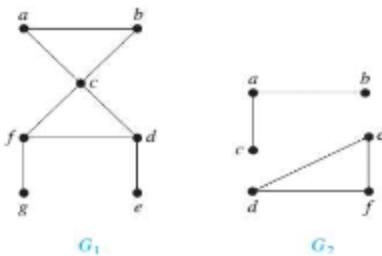
### Example (28)

Figure (a) and (b) are connected,  
however (c) and (d) are disconnected.



### Example (29)

Are the following graphs connected?



The answer

The Graphs  $G_1$  and  $G_2$

The graph  $G_1$  is **connected**, because for every pair of distinct vertices **there is a path** between them.

However, the graph  $G_2$  is **not connected**.

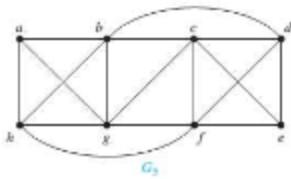
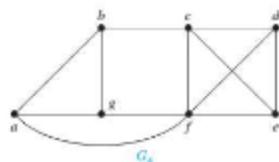
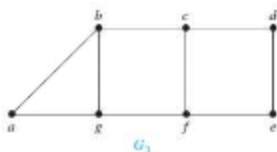
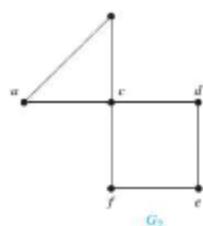
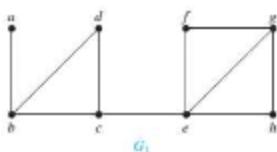
For instance, there is no path in  $G_2$  between vertices  $a$  and  $d$ .

## Theorem

There is a simple path between every pair of distinct vertices of a **connected undirected** graph.

## Example (30)

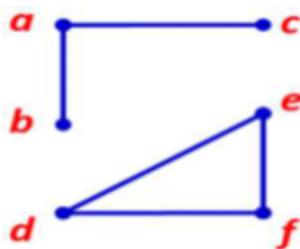
Some Connected Graphs are shown in the below figure:



Some Connected Graphs

## Connected Component

A graph that is disconnected is the union of two or more disjoint connected subgraphs called the **connected components** of the graph.

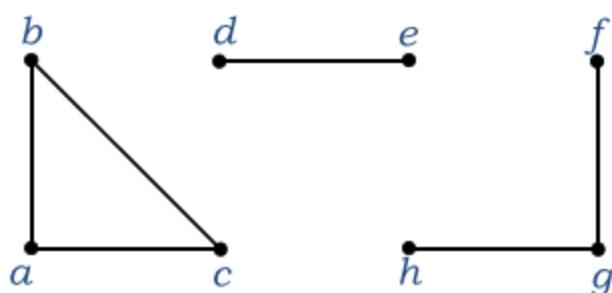


A graph  $G$  that is not connected has two or more connected components that are disjoint and have  $G$  as their union.

23

### Example (31)

What are the connected components of the following graph  $H$ ?



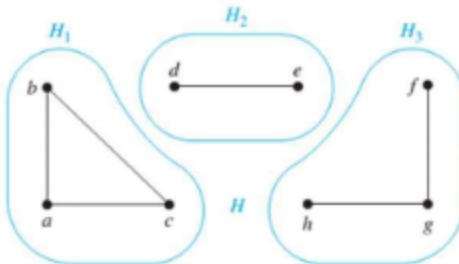
The Graphs  $H$

24

## Example (31)

### Solution:

The graph  $H$  is the union of three disjoint connected subgraphs  $H_1$ ,  $H_2$ , and  $H_3$ , shown in the following Figure. These three subgraphs are the connected components of  $H$ .



The Graph  $H$  and its Connected Components  $H_1$ ,  $H_2$ , and  $H_3$ .

25

## Cut vertex

### How Connected is a Graph?

- Sometimes the **removal** of a vertex and all incident edges from a graph produces a subgraph with more connected components.
- Such vertices are called **cut vertices** (or **articulation points**).
- The removal of a **cut vertex** from a connected graph produces a subgraph that is **not connected**.

26

## Cut / Bridge Edges

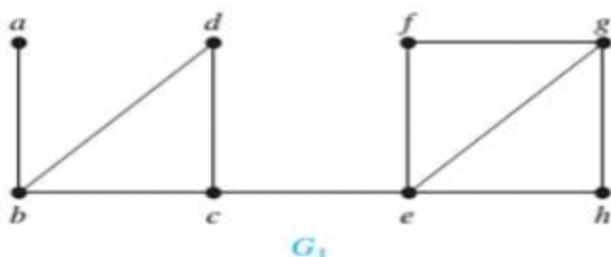
- An edge whose removal produces a graph with more connected components than in the original graph is called a **cut edge or bridge**.
- Note that in a graph representing a computer network, a **cut vertex and a cut edge** represent an **essential router** and an **essential link** that cannot fail for all computers to be able to communicate.

A **cut edge (bridge)** is an edge that if removed from a connected graph would leave behind a disconnected graph.

27

### Example (32)

Find the **cut vertices** and **cut edges** in the graph  $G_1$  shown in following Figure.



#### Solution:

The **cut vertices** of  $G_1$  are **b**, **c**, and **e**. The removal of one of these vertices (and its adjacent edges) disconnects the graph.

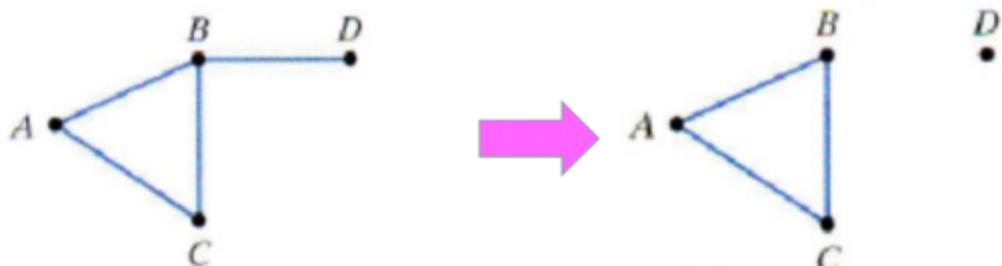
The **cut edges** are  $\{a, b\}$  and  $\{c, e\}$ . Removing either one of these edges disconnects  $G_1$ .

28

## Example (33)

In the below figure, if edge BD were removed then vertex D would be isolated from the rest of the graph, leaving behind a disconnected graph.

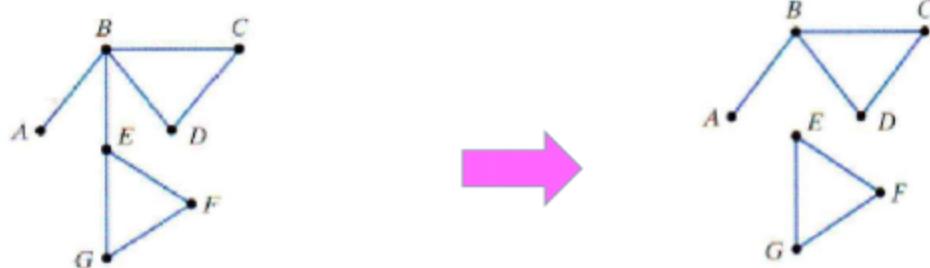
Thus, BD is a **bridge** for the given graph.



29

## Example (34)

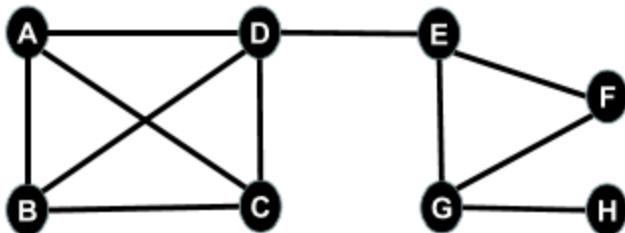
- If an edge BE were removed from the Figure shown below, the graph, leaving behind become a disconnected graph. As shown by the resulting two separated components. Thus, BE is a bridge for the given graph.



An edge is a cut edge (bridge) when the removal of this edge produces a subgraph that is not connected.

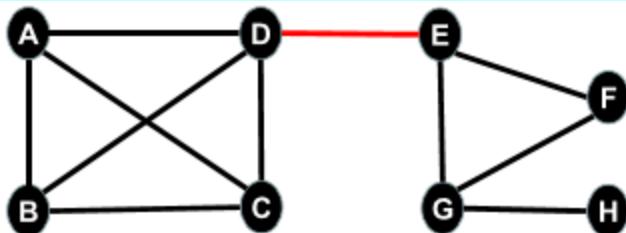
## Example (35)

The below graph has only one component, what are edges whose removal would disconnect the graph?



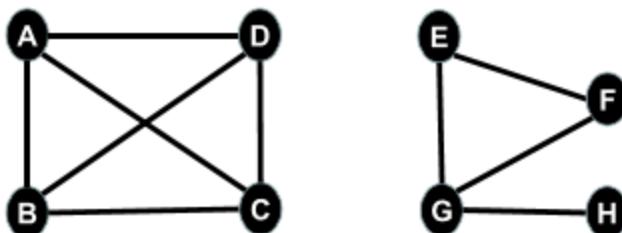
31

## Example (35)



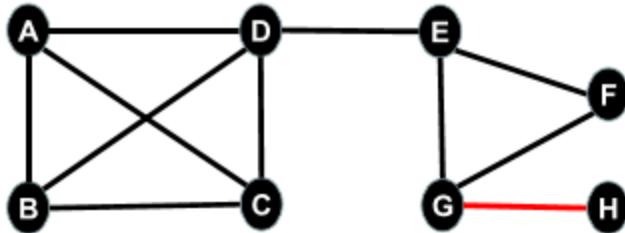
The answer

DE is a cut edge; because if it is removed the graph become disconnect the graph is divided into two components, obtaining graph (a).

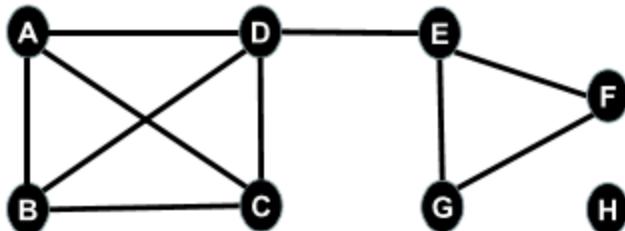


32

## Example (35)

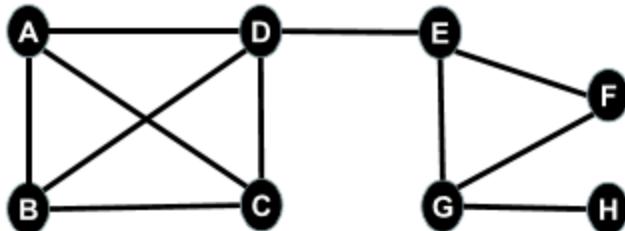


- HG is a cut edge; if we removed this edge, we would disconnect the graph into two components (one of which would be the single vertex H), obtaining graph (b)



33

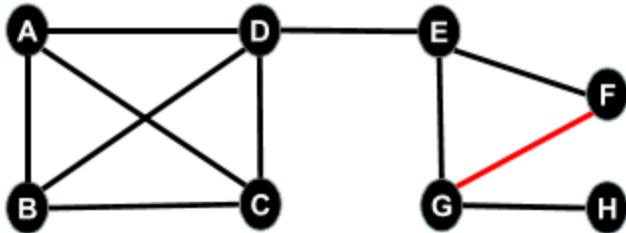
## Example (35)



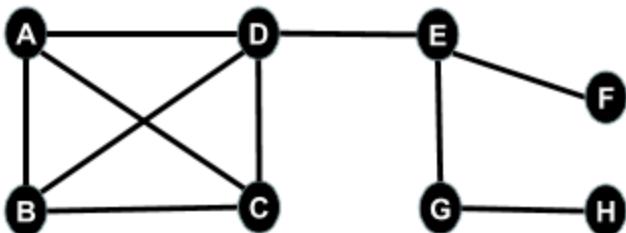
- None of the other edges is a cut edge; we could remove anyone of the other edges and still have a connected graph (that is, we would still have a path from each vertex of the graph to each other vertex).

34

## Example (35)



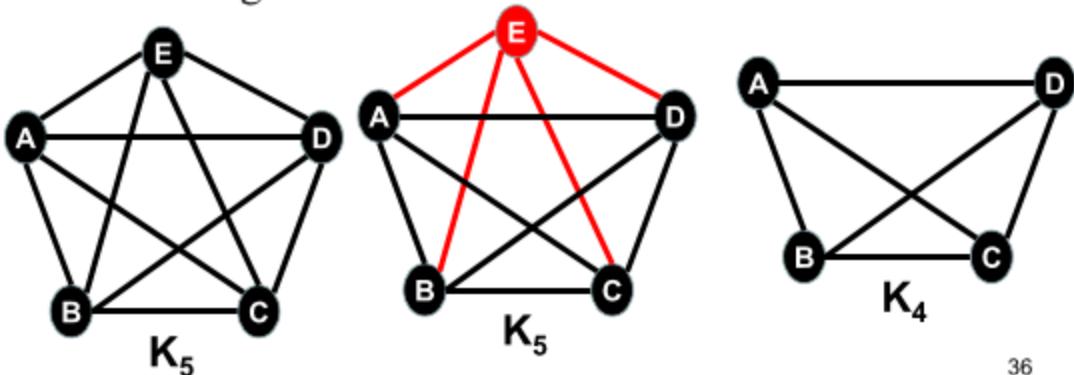
- For example, if we remove edge GF, we end up with a graph that still is connected.



35

## Vertex Connectivity

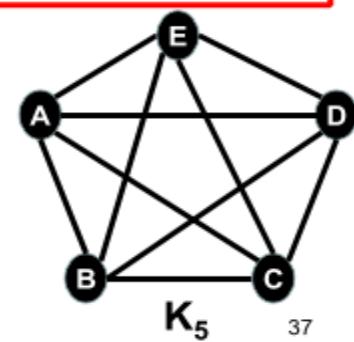
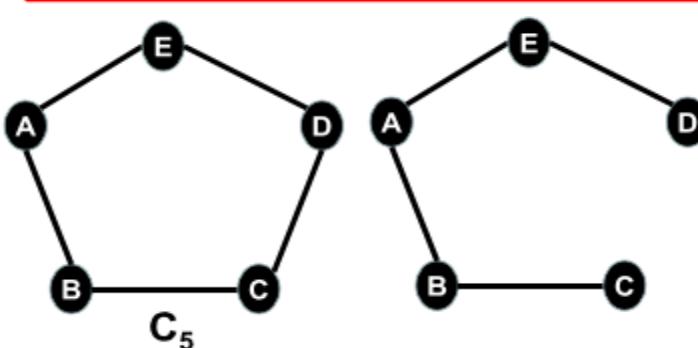
- Not all graphs have **cut vertices**.
- The complete graph  $K_n$ , where  $n \geq 3$ , has no cut vertices.
- When you remove a vertex from  $K_n$  and all edges incident to it, the resulting subgraph is the complete graph  $K_{n-1}$ , a connected graph, such as  $K_5$  as shown in the below figure.



36

## Vertex Connectivity

- Connected graphs without cut vertices are called **non-separable graphs** and can be thought of as **more connected** than those with a **cut vertex**.
- Defining a more granulated measure of graph connectivity based on the **minimum number of vertices** that can be removed to **disconnect** a graph.



37

## Representing Graphs on Computers

### 1. Adjacency Matrix for Simple Graph

- Undirected Graph
- Directed Graph

### 2. Representation Matrix for Not Simple Graph

- Undirected Graph
- Directed Graph

### 3. Drawing a Graph from Adjacency Matrix

5

Adjacency Matrices

## Adjacency Matrices for Simple Undirected Graph

### Adjacency Matrix for Simple Undirected Graph

- In graph theory and computer science, an **adjacency matrix** is a square matrix used to represent whether pairs of vertices **are adjacent or not** in the graph.

## Adjacency Matrix (Simple Undirected Graph)

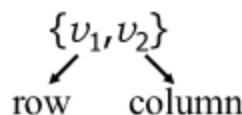
- Suppose that  $G = (V, E)$  is a **simple graph** where  $|V|=n$ .
- Suppose that the vertices of  $G$  are listed arbitrarily as  $v_1, v_2, \dots, v_n$ .
- The adjacency matrix  $A$  of the **simple graph**  $G$ , with respect to this listing of the vertices, is the  $n \times n$  **(0,1)-matrix** with
  - 1 as its  $(i, j)^{\text{th}}$  entry when  $v_i$  and  $v_j$  **are adjacent**, and
  - 0 as its  $(i, j)^{\text{th}}$  entry when they **are not adjacent**.
  - zeros on its diagonal (Because it is simple graph → No loop )

9

## Adjacency Matrix (Simple Undirected Graph)

- A **simple graph**  $G = (V, E)$  with  $n$  vertices can be represented by an **adjacency matrix A**, where the entry  $a_{ij}$  in row  $i$  and column  $j$  is:

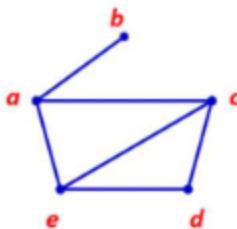
$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge in } G \\ 0 & \text{otherwise} \end{cases}$$



10

## Example (36)

Use adjacency matrix to represent the graph below.



$$\mathbf{A} = \begin{bmatrix} a & b & c & d & e \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

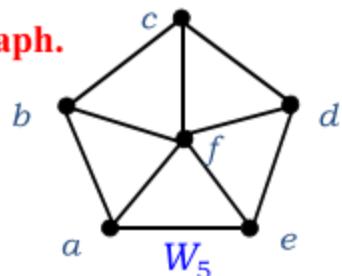
$\{v_1, v_2\}$   
row      column

Note that  $\mathbf{A}$  is symmetric

11

## Example (37)

Use an adjacency matrix to represent this graph.



Solution:

We order the vertices as a, b, c, d, e. The matrix representing this graph is

From	To	$a$	$b$	$c$	$d$	$e$	$f$
$a$		0	1	0	0	1	1
$b$		1	0	1	0	0	1
$c$		0	1	0	1	0	1
$d$		0	0	1	0	1	1
$e$		1	0	0	1	0	1
$f$		1	1	1	1	1	0

12

## Adjacency Matrices for Simple Directed Graph

### Directed graphs

- Directed graphs can be represented by **zero-one** matrices.
- The matrix for a directed graph  $G = (V, E)$  has a 1 in its  $(i, j)^{\text{th}}$  position if there is an edge from  $V_i$  to  $V_j$ , where  $V_1, V_2, \dots, V_n$  is an arbitrary listing of the vertices of the directed graph.

## Directed graphs

- In other words, if  $A = [a_{ij}]$  is the adjacency matrix for the directed graph with respect to this listing of the vertices, then

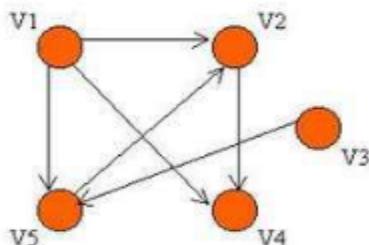
$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge in } G \\ 0 & \text{otherwise} \end{cases}$$

- The adjacency matrix for a directed graph **does not have to be symmetric**, because there may not be an edge from  $v_j$  to  $v_i$  when there is an edge from  $v_i$  to  $v_j$ .

15

## Example (38)

Use an adjacency matrix to represent this directed graph.



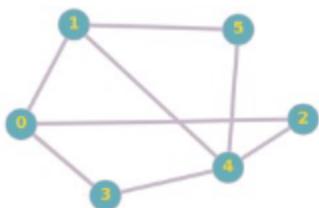
Solution:

	v1	v2	v3	v4	v5
v1	0	1	0	1	1
v2	0	0	0	1	0
v3	0	0	0	0	1
v4	0	0	0	0	0
v5	0	1	0	0	0

16

## Example (39)

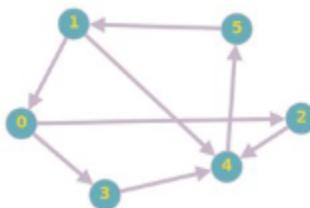
Use an adjacency matrix to represent these two graphs.



	0	1	2	3	4	5
0	0	1	1	1	0	0
1	1	0	0	0	1	1
2	1	0	0	0	1	0
3	1	0	0	0	1	0
4	0	1	1	1	0	1
5	0	1	0	0	1	0

Adjacency Matrix for Undirected Graph

symmetric



Solution:

	0	1	2	3	4	5
0	0	0	1	1	0	0
1	1	0	0	0	1	0
2	0	0	0	0	1	0
3	0	0	0	0	1	0
4	0	0	0	0	0	1
5	0	1	0	0	0	0

Adjacency Matrix for Directed Graph

not symmetric

17

## Note on simple graph

- The adjacency matrix of a **simple graph** is symmetric, that is,  $a_{ij} = a_{ji}$ , because both of these entries are 1 when  $V_i$  and  $V_j$  are adjacent, and both are 0 otherwise.
- Furthermore, because a simple graph has no loops, each entry  $a_{ii}$ ,  $i = 1, 2, 3, \dots, n$ , is 0

18

## Sparse graph vs Dense graph

### ➤ Sparse graph

- When a simple graph contains **relatively few edges**, that is, when it is **sparse**.
- Note, the adjacency matrix of a sparse graph is a **sparse matrix**, that is, a matrix with few nonzero entries.

### ➤ Dense graph

- Suppose that a simple graph is **dense**, that is, suppose that it contains many edges, such as a graph that contains more than **half** of all possible edges.

Representation Matrix

## Representation Matrix for Not Simple Undirected Graph

### Representation Matrix (not Simple Undirected Graph)

- Adjacency matrices can also be used to represent **undirected graphs** with **loops** and with **multiple** edges.
- A loop at the vertex  $v_i$  is represented by a 1 at the  $(i, i)^{th}$  position of the adjacency matrix.
- **When multiple edges are present**, the adjacency matrix is **no longer a zero-one matrix**, because  $(i, j)^{th}$  entry of this matrix equals the **number of edges** that are associated to  $\{v_i, v_j\}$ .

## Representation Matrix (not Simple Undirected Graph)

- The adjacency matrix of  $G$  is the  $n \times n$  matrix  $A = (a_{ij})$  where

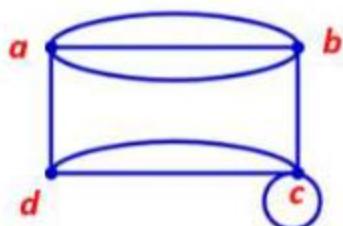
$a_{ij} = \text{the number of edges connecting } v_i \text{ and } v_j$   
for all  $i, j = 1, 2, \dots, n$

All undirected graphs, including multi-graphs and pseudo-graphs, have **symmetric adjacency matrices**.

23

## Example (40)

Use representation matrix to represent the graph below.



Solution:

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

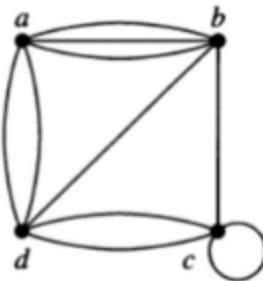
$a \quad b \quad c \quad d$

$a \quad b \quad c \quad d$

24

## Example (41)

Use representation matrix to represent the pseudo-graph below.



Solution:

The representation matrix using the ordering of vertices a, b, c, d is

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

25

Representation Matrix for Not Simple Directed Graph

## Directed multigraphs

- Adjacency matrices can also be used to represent directed **multigraphs**.
- Again, such matrices **are not zero-one matrices** when there are **multiple** edges in the same direction connecting two vertices.

27

## Directed multigraphs

- In the adjacency matrix for a directed multigraph,  $a_{ij}$  equals the **number of edges that are associated to**  $(V_i, V_j)$ .

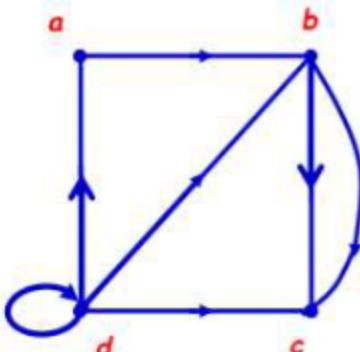
$$a_{ij} = \text{the number of edges from } v_i \text{ to } v_j \\ \text{for all } i, j = 1, 2, \dots, n$$

The matrix is not symmetric.

28

## Example (42)

Use Representation matrix to represent the graph below.



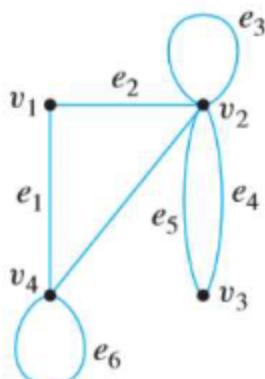
Solution:

$$\begin{array}{l} \begin{matrix} & a & b & c & d \\ a & \left[ \begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \right] \\ b & \\ c & \\ d & \end{matrix} \end{array}$$

29

## Example (43)

Find the Representation matrix for the graph G shown below.



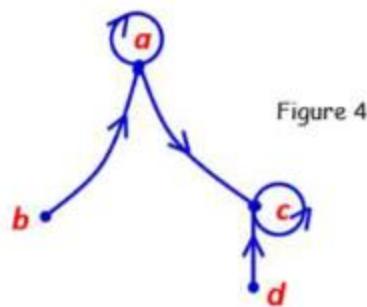
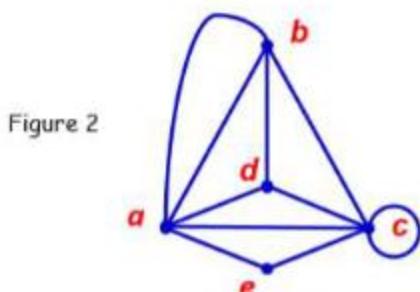
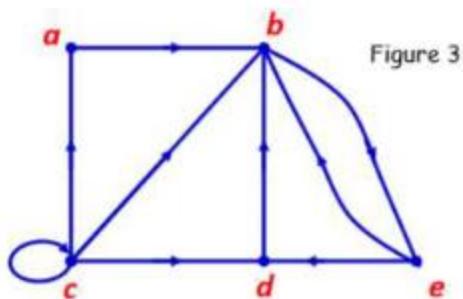
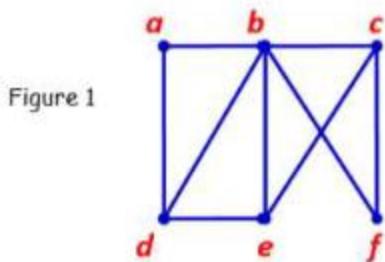
Solution:

$$A = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & \left[ \begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{matrix} \right] \\ v_2 & \\ v_3 & \\ v_4 & \end{bmatrix}$$

30

## Exercise (5): check your understanding

Represent the graph with a representation matrix



31

Drawing a Graph from Adjacency Matrix

## Drawing a Graph from Adjacency Matrix

- An adjacency matrix of a graph is based on the ordering chosen for the vertices.
- There are  $n!$  different adjacency matrices for a graph with  $n$  vertices, because there are  $n!$  different ordering of  $n$  vertices.

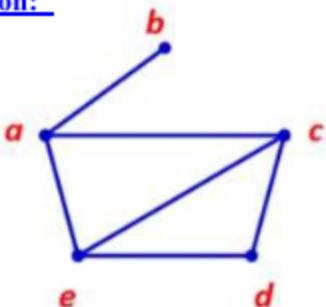
33

## Example (44)

Draw a graph with the given adjacency matrix

$$A = \begin{bmatrix} a & b & c & d & e \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \begin{array}{l} a \\ b \\ c \\ d \\ e \end{array}$$

Solution:



Symmetric → undirected graph or  
directed symmetric graph

34

## Example (45)

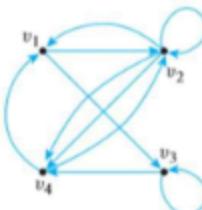
Draw a directed graph that has A as its adjacency matrix.

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

### Solution

Let G be the graph corresponding to A, and let  $v_1, v_2, v_3, v_4$  be the vertices of G. Label A across the top and down the left side with these vertex names, as shown below.

$$A = \begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 0 & 1 & 1 & 0 \\ v_2 & 1 & 1 & 0 & 2 \\ v_3 & 0 & 0 & 1 & 1 \\ v_4 & 2 & 1 & 0 & 0 \end{array}$$



- Then, for instance, the 2 in the fourth row and the first column means that there are **two** arrows from  $v_4$  to  $v_1$ .
- The 0 in the first row and the fourth column means that there is no arrow from  $v_1$  to  $v_4$ . A corresponding directed graph is shown on the RHS figure.

35

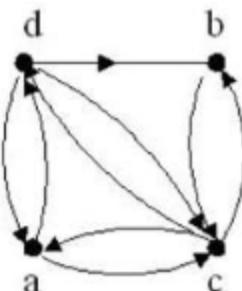
## Example (46)

Draw a graph with the given adjacency matrix.

$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 0 & 1 & 1 \\ b & 0 & 0 & 1 & 0 \\ c & 1 & 1 & 0 & 1 \\ d & 1 & 1 & 1 & 0 \end{array}$$

### Solution:

Not symmetric → must directed graph



36

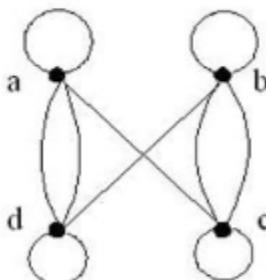
## Example (47)

Draw the **graph** representing the following adjacency matrix.

$$\begin{array}{l} a \quad b \quad c \quad d \\ \hline a & \left[ \begin{array}{cccc} 1 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{array} \right] \\ b \\ c \\ d \end{array}$$

Solution:

Order the vertices as a, b, c, d.



37

## Exercise (6): check your understanding

Draw a graph with the given matrix

a)  $\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$

Adjacency matrix

b)  $\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$

Representation matrix

c)  $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

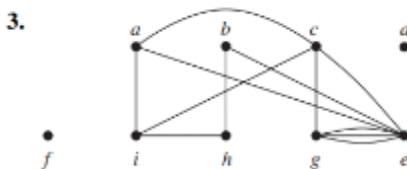
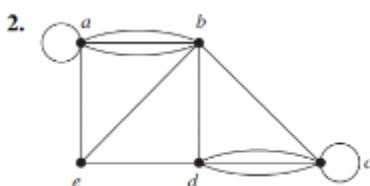
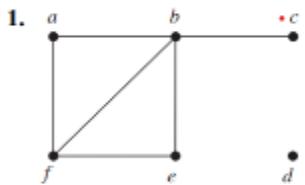
Representation matrix

38

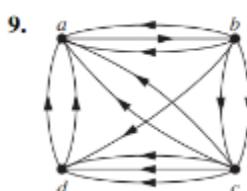
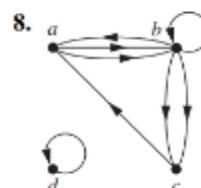
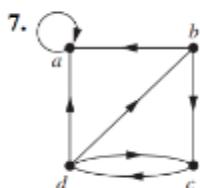
## Section No. 01

Answer the following Questions.

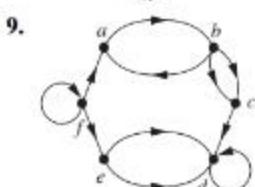
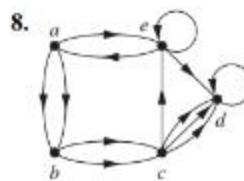
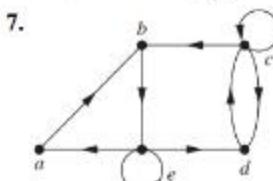
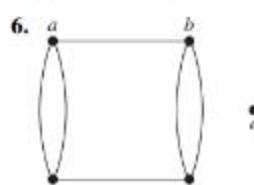
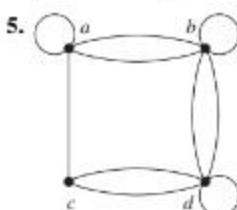
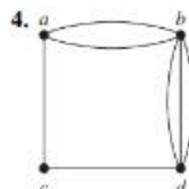
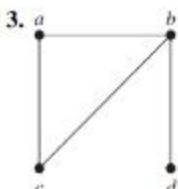
1. In Exercises 1–3 find the number of vertices, the number of edges, and the degree of each vertex in the given undirected graph. Identify all isolated and pendant vertices.



2. Find the sum of the degrees of the vertices of each graph in Exercises 1–3 and verify that it equals twice the number of edges in the graph.
3. In Exercises 7–9 determine the number of vertices and edges and find the in-degree and out-degree of each vertex for the given directed multigraph.



4. For each of the graphs in Exercises 7–9 determine the sum of the in-degrees of the vertices and the sum of the out-degrees of the vertices directly. Show that they are both equal to the number of edges in the graph.
5. For Exercises 3–9, determine whether the graph shown has directed or undirected edges, whether it has multiple edges, and whether it has one or more loops. Use your answers to determine the type of graph in Table 1 this graph is.



6. For each undirected graph in Exercises 3–9 that is not simple, find a set of edges to remove to make it simple.

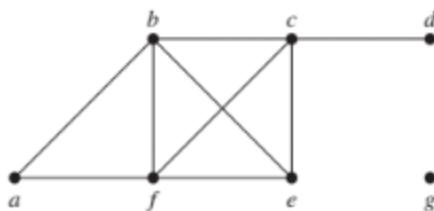
**TABLE 1** Graph Terminology.

Type	Edges	Multiple Edges Allowed?	Loops Allowed?
Simple graph	Undirected	No	No
Multigraph	Undirected	Yes	No
Pseudograph	Undirected	Yes	Yes
Simple directed graph	Directed	No	No
Directed multigraph	Directed	Yes	Yes
Mixed graph	Directed and undirected	Yes	Yes

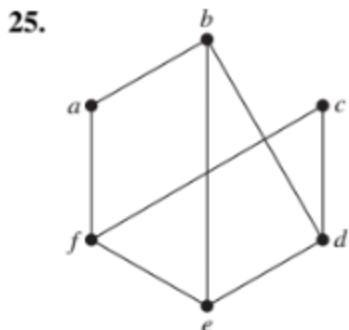
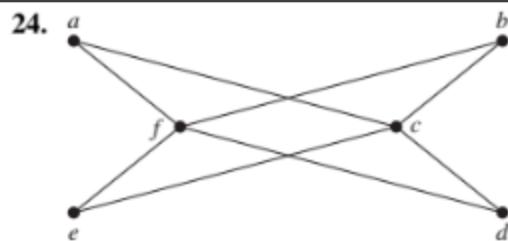
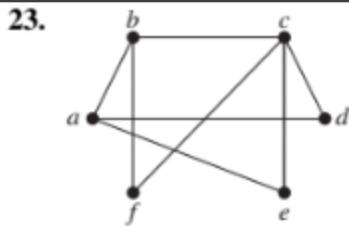
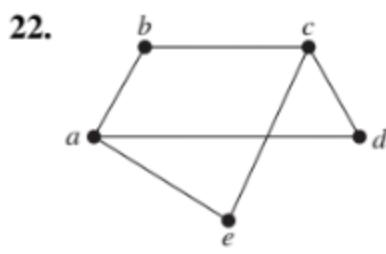
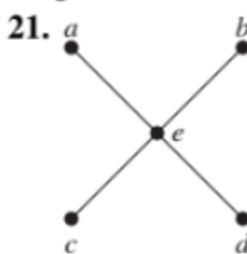
## Section No. 02

Answer the following Questions.

1. The degree sequence of a graph is the sequence of the degrees of the vertices of the graph in no increasing order. For example, the degree sequence of the following graph G is 4, 4, 4, 3, 2, 1, 0.



2. Find the degree sequences for each of the graphs in Exercises 21–25.

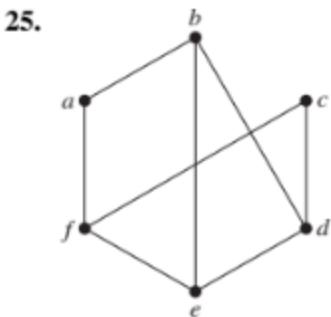
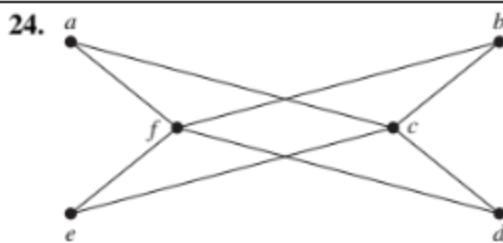
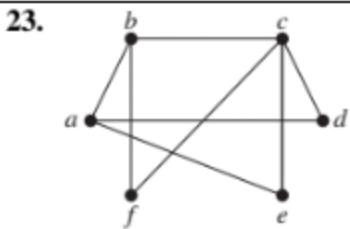
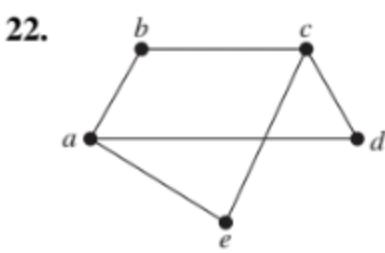
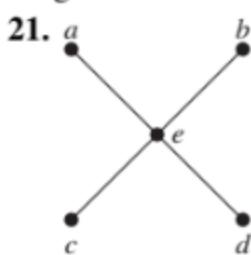


3. Find the degree sequence of each of the following graphs.

- a)  $K_4$
- b)  $C_4$
- c)  $W_4$
- d)  $K_{2,3}$
- e)  $Q_3$

4. How many edges does a graph have if its degree sequence is 4, 3, 3, 2, 2? Draw such a graph.

5. In Exercises 21–25 determine whether the graph is bipartite.



6. Draw these graphs.

- a)  $K_7$       b)  $K_{1,8}$       c)  $K_{4,4}$   
d)  $C_7$       e)  $W_7$       f)  $Q_4$

7. For which values of  $n$  are these graphs bipartite?

- a)  $K_n$       b)  $C_n$       c)  $W_n$       d)  $Q_n$

8. How many vertices and how many edges do these graphs have?

- a)  $K_n$       b)  $C_n$       c)  $W_n$   
d)  $K_{m,n}$       e)  $Q_n$

9. For which values of  $n$  are these graphs regular?

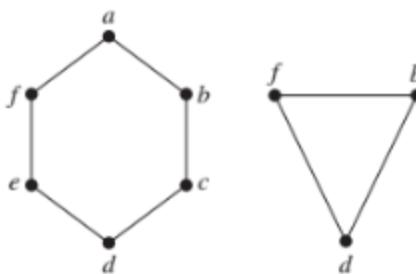
- a)  $K_n$
- b)  $C_n$
- c)  $W_n$
- d)  $Q_n$

10. For which values of  $m$  and  $n$  is  $K_{m,n}$  regular?

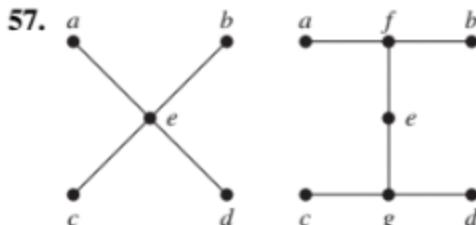
11. How many vertices does a regular graph of degree four with 10 edges have?

12. In Exercises 56–58 find the union of the given pair of simple graphs. (Assume edges with the same endpoints are the same.)

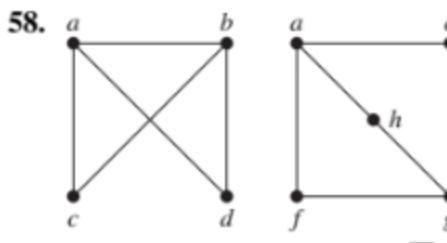
56.



57.



58.

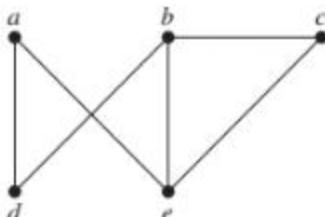


## Section No. 03

### Answer the following Questions.

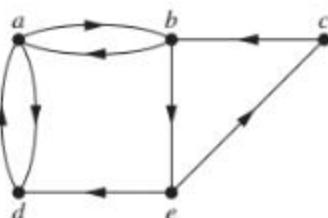
1. Does each of these lists of vertices form a path in the following graph? Which paths are simple? Which are circuits? What are the lengths of those that are paths?

- a)  $a, e, b, c, b$       b)  $a, e, a, d, b, c, a$   
c)  $e, b, a, d, b, e$       d)  $c, b, d, a, e, c$



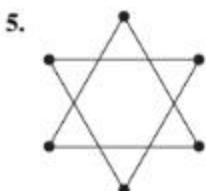
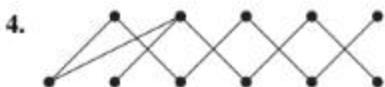
2. Does each of these lists of vertices form a path in the following graph? Which paths are simple? Which are circuits? What are the lengths of those that are paths?

- a)  $a, b, e, c, b$       b)  $a, d, a, d, a$   
c)  $a, d, b, e, a$       d)  $a, b, e, c, b, d, a$



3. In Exercises 3–5

- a) determine whether the given graph is connected.  
b) How many connected components does each of the graphs in Exercises 3–5 have?  
c) For each graph find each of its connected components.

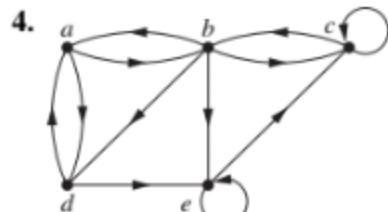
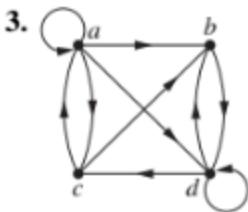
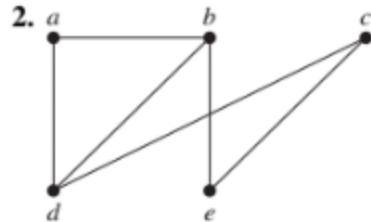
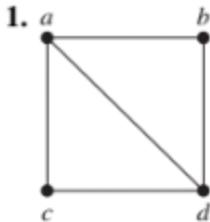


4. Show that each of the following graphs

- a) has no cut vertices.
- b) has no cut edges.

- a)  $C_n$  where  $n \geq 3$
- b)  $W_n$  where  $n \geq 3$
- c)  $K_{m,n}$  where  $m \geq 2$  and  $n \geq 2$
- d)  $Q_n$  where  $n \geq 2$

5. Represent the graph in Exercise 1 with an adjacency matrix.



6. Represent each of these graphs with an adjacency matrix.

- a)  $K_4$
- b)  $K_{1,4}$
- c)  $K_{2,3}$
- d)  $C_4$
- e)  $W_4$
- f)  $Q_3$

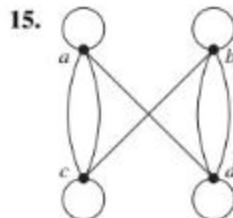
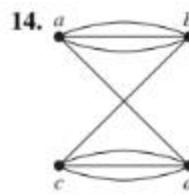
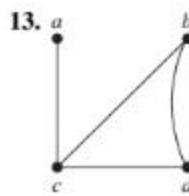
7. In Exercises 10–12 draw a graph with the given adjacency matrix

10. 
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

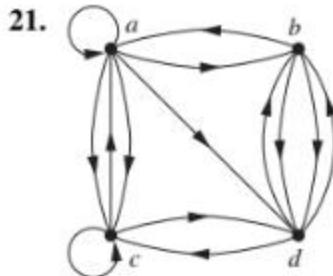
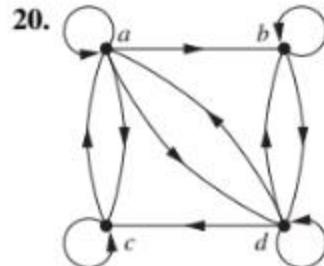
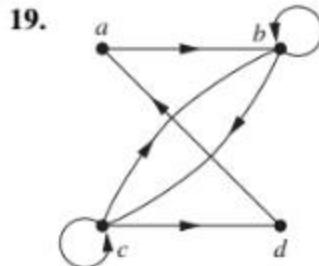
11. 
$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

12. 
$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

8. In Exercises 13–15 represent the given graph using an adjacency matrix.



9. In Exercises 19–21 find the adjacency matrix of the given directed multigraph with respect to the vertices listed in alphabetic order.



10. In Exercises 22–24 draw the graph represented by the given adjacency matrix.

22.  $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$     23.  $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 0 \\ 0 & 2 & 2 \end{bmatrix}$     24.  $\begin{bmatrix} 0 & 2 & 3 & 0 \\ 1 & 2 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix}$

11. Find an adjacency matrix for each of these graphs.

- a)  $K_n$     b)  $C_n$     c)  $W_n$     d)  $K_{m,n}$     e)  $Q_n$

# **Chapter (5)**

## **Trees**

# Key Terms and Overview

## Introduction to Trees

Tree

Forest

rooted tree

## The terminology for trees

Subtree

child of a vertex  $v$  in a rooted tree

sibling of a vertex  $v$  in a rooted tree

ancestor of a vertex  $v$  in a rooted tree

descendant of a vertex  $v$  in a rooted tree

internal vertex

leaf

## Properties of Trees

level of a vertex

height of a tree

$m$ -ary tree

full  $m$ -ary tree

binary tree

ordered tree

5

# tree

## Definition:

A tree is a connected undirected graph with no simple circuits.

1. connected undirected graph
2. no simple circuits

## A tree is a graph that is connected and has no circuits.

All trees have the following properties:

1. There is one and only one path joining any two vertices.
2. Every edge is a bridge.
3. A tree with  $n$  vertices must have  $n - 1$  edges.

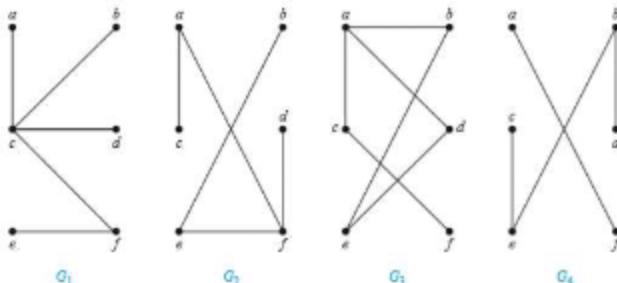
## Note that:

Because a tree cannot have a simple circuit, a tree cannot contain multiple edges or loops. Therefore any tree must be a **simple** graph.

6

# Example (1)

Which of the graphs shown in the following Figure are trees?



## Examples of Trees and graphs that are not trees

### Solution:

- $G_1$  and  $G_2$  are trees, because both are connected graphs with no simple circuits.
- $G_3$  is not a tree because  $e, b, a, d, e$  is a simple circuit in this graph.
- Finally,  $G_4$  is not a tree because it is not connected.

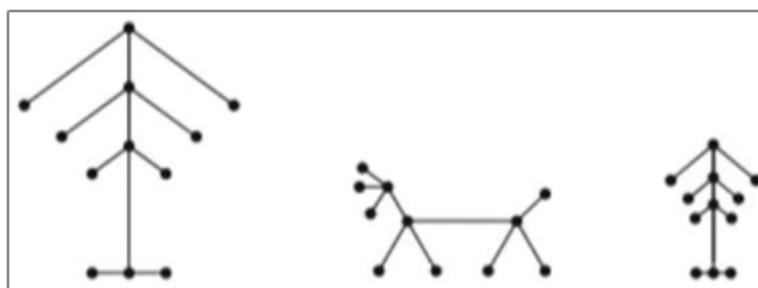
7

# Forest

### Definition:

A graph that has no simple circuits but is not necessarily connected is called a forest. The property of forest is that each of their connected components is a tree.

The below figure displays an example of a forest.

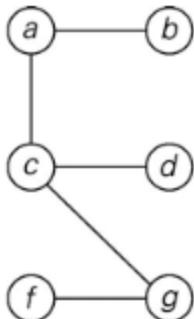


Example of a Forest.

8

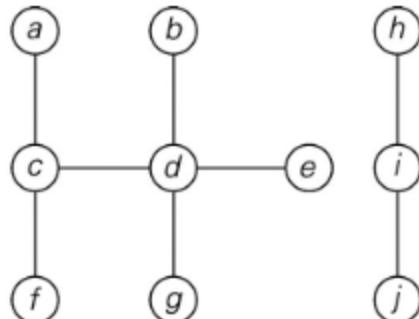
## Example (2)

Which of the graph shown in the following Figure is tree and which is forest?



(a)

A tree



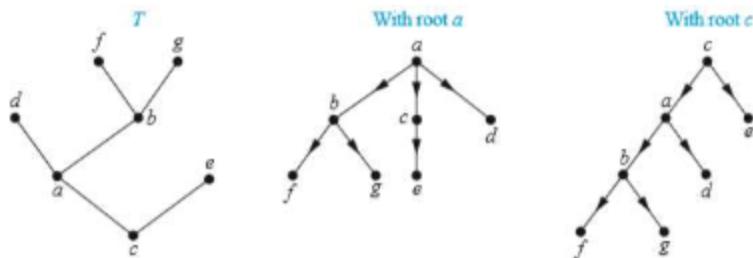
(b)

A forest

9

## Rooted Trees

- A rooted tree is a tree in which one vertex has been designated as the **root** and every edge is directed away from the root.
- We can change an **unrooted** tree into a rooted tree by choosing any vertex as the root. Note that different choices of the root produce different **rooted** trees.
- For instance, this Figure displays the rooted trees formed by designating a to be the root and c to be the root, respectively, in the tree T .



A Tree and Rooted Trees Formed by Designating Two Different Roots.

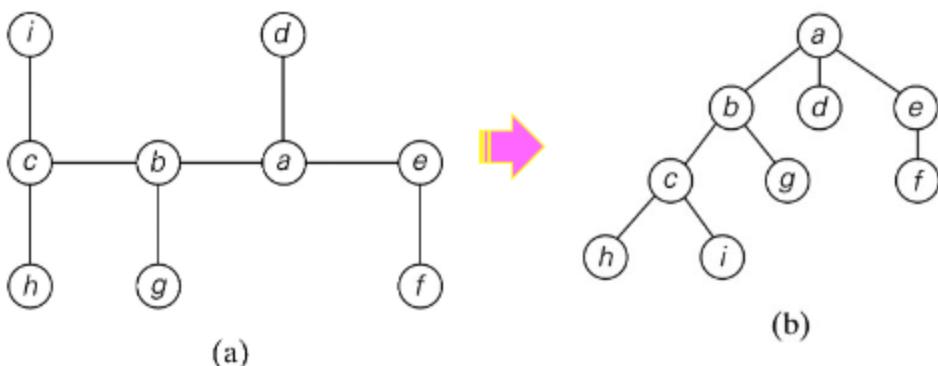
10

# Rooted Trees

- We usually draw a rooted tree with its root at the top of the graph.
- The arrows indicating the directions of the edges in a rooted tree can be omitted, because the choice of root determines the **directions** of the edges.

## Example (3)

Transform the following free tree into a rooted tree.



Transformation of a free tree into a rooted tree

# The terminology for trees

Suppose that  $T$  is a rooted tree.

➤ **Ancestors** (السلف — الاباء— الاجداد — اباء الاجداد .... )

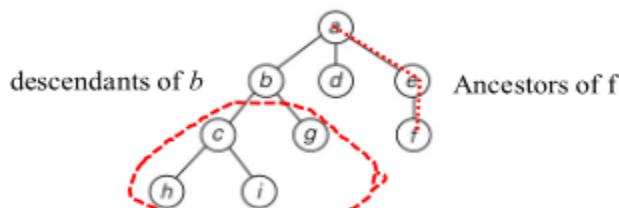
For any vertex  $v$  in a tree  $T$ , all the vertices on the simple path from the root to that vertex are called ancestors. (ancestors of  $f$  are  $e$  and  $a$ )

➤ **Descendants** (الخلف— الابناء— الاحفاد — ابناء الاحفاد .... )

All the vertices for which a vertex  $v$  is an ancestor are said to be descendants of  $v$ . (descendants of  $b$  are  $c$ ,  $g$ ,  $h$ , and  $i$ .)

➤ **parent, child and siblings** (الاخوه)

If  $(u, v)$  is the last edge of the simple path from the root to vertex  $v$  (and  $u \neq v$ ),  $u$  is said to be the **parent** of  $v$  and  $v$  is called a **child** of  $u$ . Vertices that have the same parent are called **siblings**.



13

# The terminology for trees

➤ **Leaves**

A vertex without children is called a leaf.

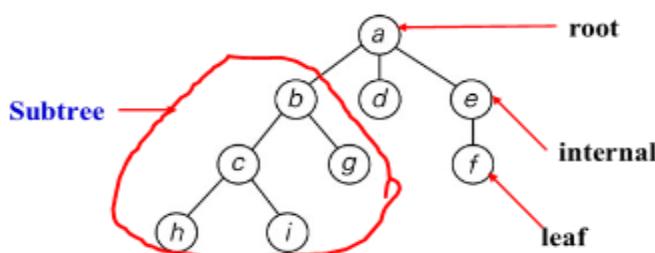
➤ **internal**

Vertices that have children are called **internal** vertices.

➤ **Subtree**

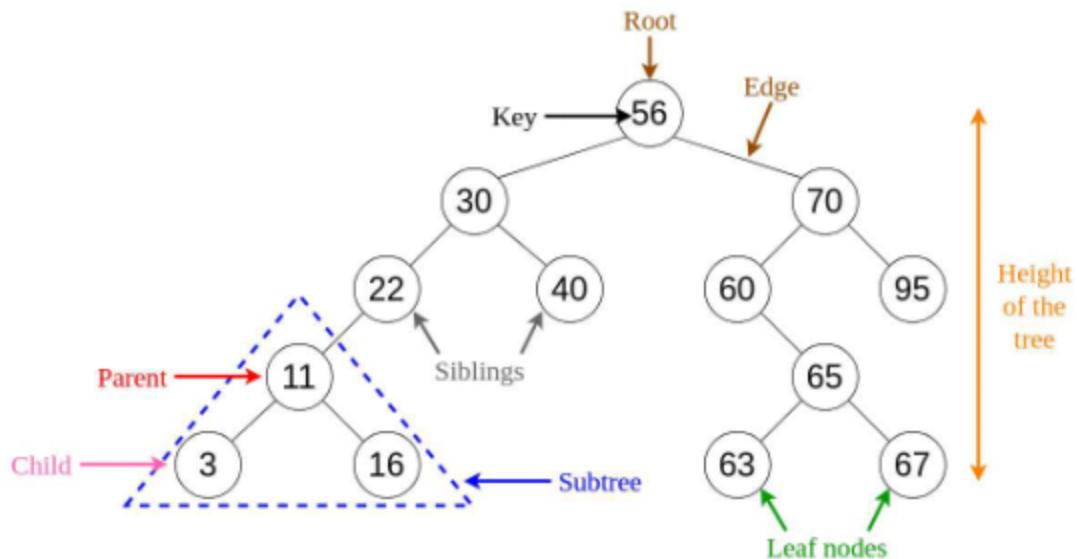
A vertex  $v$  with all its descendants is called the subtree of  $T$  rooted at  $v$ .

**Note that :** The **root** is an **internal** vertex unless it is the only vertex in the graph, in which case it is a **leaf**.



14

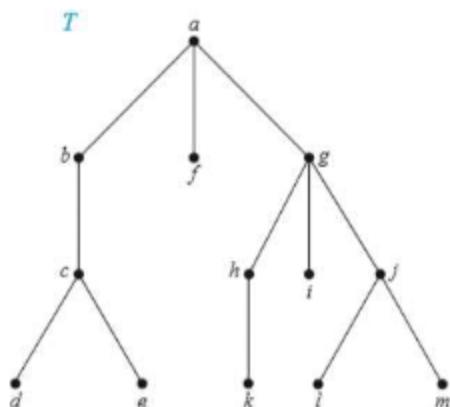
# Basic Terminology of Trees



15

## Example (4)

In the rooted tree T (with root a) shown in the Figure, find :

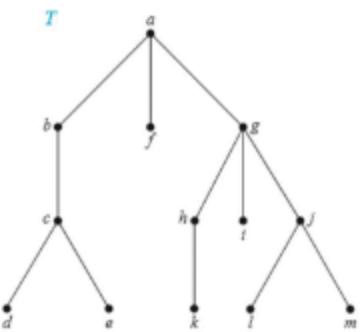


1. The parent of c,
2. The children of g,
3. The siblings of h,
4. All ancestors of e,
5. All descendants of b,
6. All internal vertices,
7. All leaves.
8. What is the subtree rooted at g?

A Rooted Tree T

16

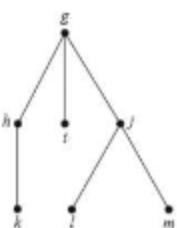
## Example (4)



Solution:

1. The parent of c is b.
2. The children of g are h, i, and j.
3. The siblings of h are i and j.
4. The ancestors of e are c, b, and a.
5. The descendants of b are c, d, and e.
6. The internal vertices are a, b, c, g, h, and j.
7. The leaves are d, e, f, i, k, l, and m.
8. The subtree rooted at g is shown in Figure 6.

A Rooted Tree T

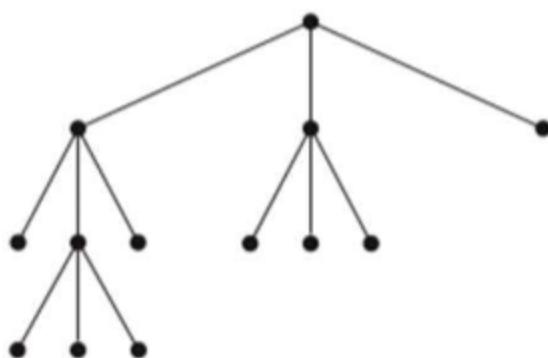


The Subtree Rooted at g.

17

## m-ary tree and full m-ary tree

- A rooted tree is called an **m-ary tree** if every **internal** vertex has **no more** than m children.
- The tree is called a **full m-ary tree** if every **internal** vertex has **exactly m children**.
- An m-ary tree with m = 2 is called a binary tree.

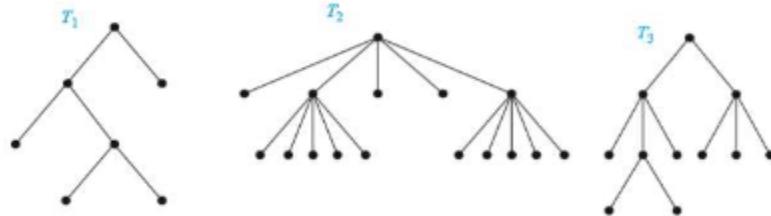


T is a **full 3-ary tree** because each of its internal vertices has three children.

18

## Example (5)

Are the rooted trees in this Figure full m-ary trees for some positive integer m?



Four Rooted Trees.

Solution:

- $T_1$  is a **full binary tree** because each of its internal vertices has two children.
- In  $T_3$  each internal vertex has five children, so  $T_3$  is a **full 5-ary tree**.
- $T_4$  is not a full m-ary tree for any m because some of its internal vertices have two children and others have three children.

19

## Representing Organizations using rooted tree

- Representing Organizations The structure of a large organization can be modeled using a rooted tree.
- Each vertex in this tree represents a position in the organization.
- An edge from one vertex to another indicates that the person represented by the initial vertex is the (direct) boss of the person represented by the terminal vertex. The graph shown in this Figure displays such a tree.



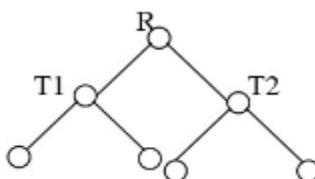
An Organizational Tree for a Computer Company.

20

# Binary tree

## Binary trees

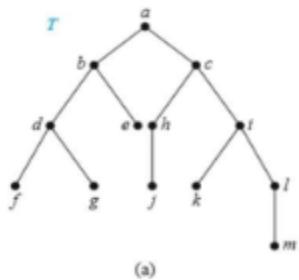
- A binary tree is an ordered tree in which every vertex has no more than two children and each child is designated as either a left child or a right child of its parent.
- In a binary tree, if an internal vertex has two children, the **first** child is called the **left** child and the second child is called the **right** child.
- The tree **rooted** at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex.



A binary tree

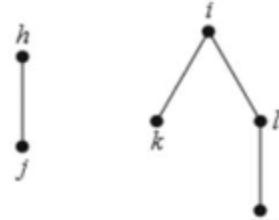
21

## Example (6)



A Binary Tree T

1. What are the left and right children of d in the binary tree T shown in this Figure?
2. What are the left and right subtrees of c?



### Solution:

The left child of d is f and the right child is g.  
The left and right subtrees of c in Figures (b) and (c), respectively.

Left and Right Subtrees  
of the Vertex c.

22

## Properties of Trees

23

## Properties of Trees

Trees have several important properties other graphs do not have.

**Theorem:** The number of edges in a tree is always one less than the number of its vertices:  $|E| = |V| - 1$   
i.e. A tree with  $n$  vertices has  $n-1$  edges.

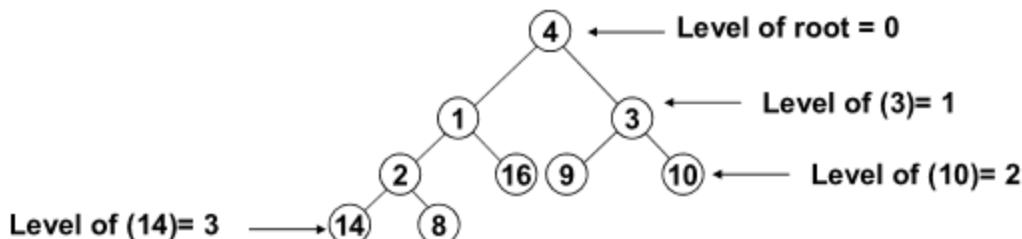
**Theorem:** For every two vertices in a tree there always exists exactly one simple path from one of these vertices to the other.

24

# Definitions: The level of a vertex

**The level of a vertex v in a tree:** is the length of the unique path from the **root** to this **vertex**.

**The level of the root is defined to be zero.**

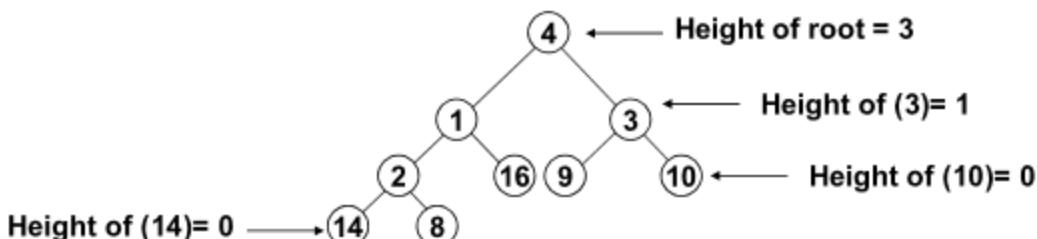


25

# Definition: Height of a vertex

**Height of a vertex v in a tree:** is the number of edges on the longest simple path from the **vertex** down to a **leaf**.

**The Height of all leaves is zero.**

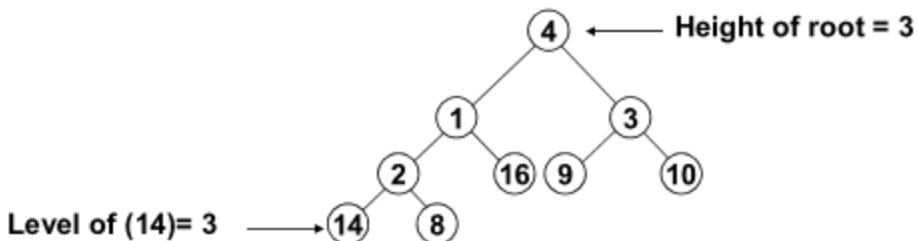


26

# Definitions: The height of a tree

**The height of a tree:** it is the length of the longest path from the root to any leaf. = It is also the height of root node

$h_t$  = the height of a tree T is length of the longest path from root to leaf

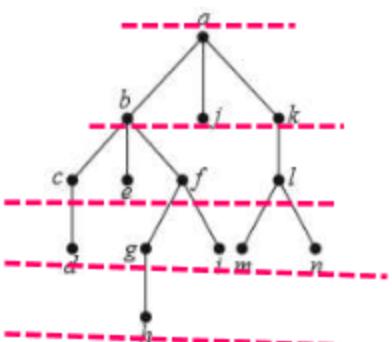


27

## EXAMPLE (7)

Find the level of each vertex in the rooted tree shown in this Figure. What is the height of this tree?

A Rooted Tree.



### Solution:

- The root a is at level 0.
- Vertices b, j, and k are at level 1.
- Vertices c, e, f, and l are at level 2.
- Vertices d, g, i, m, and n are at level 3.
- Finally, vertex h is at level 4.
- Because the largest level of any vertex is 4, this tree has height 4.

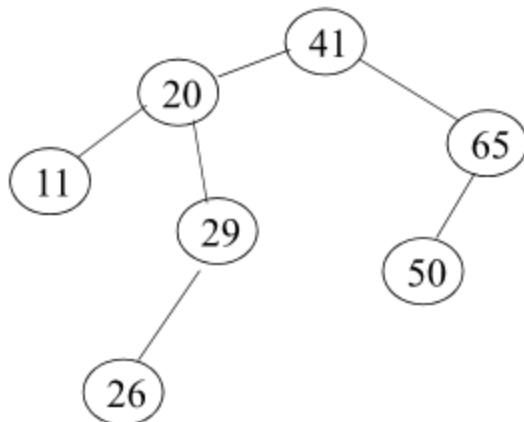
28

## Example (8)

What is the height of each vertex in this tree?

What is the level of each vertex in this tree?

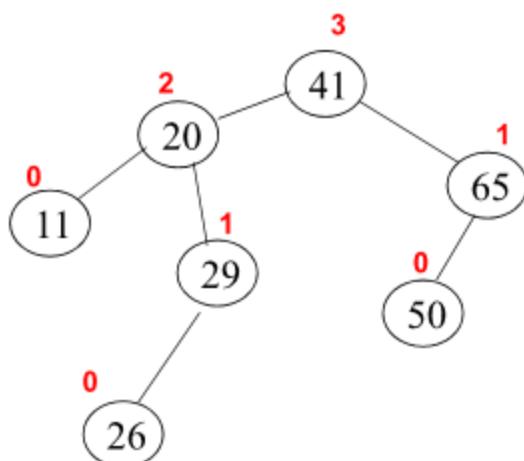
What is the height of this tree?



29

## Example (8)

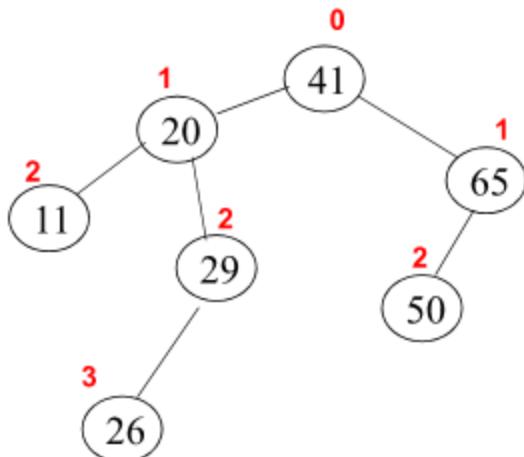
The height of each vertex in this tree is as follows:



30

## Example (8)

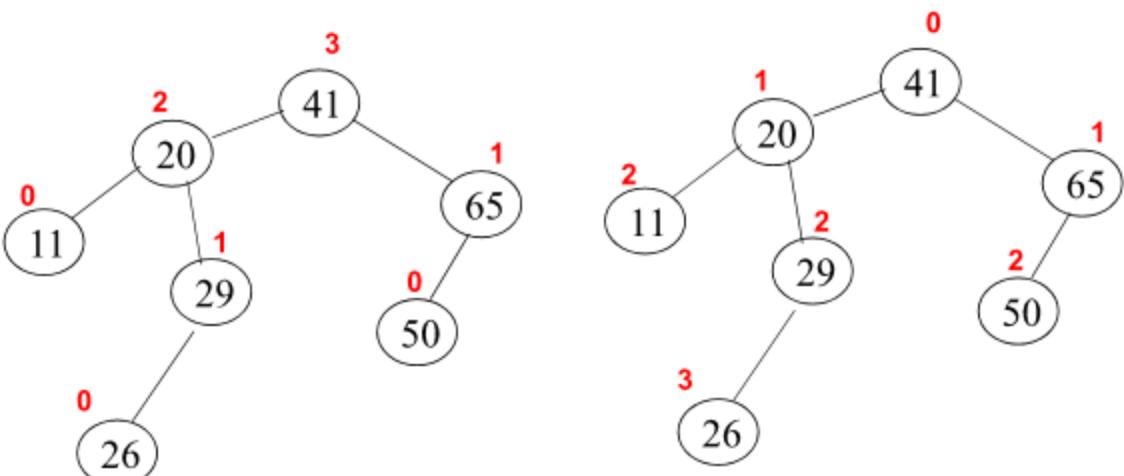
The level of each vertex in this tree is as follows:



31

## Example (8)

The height of the tree is as follows:  
 $h = \max(\text{level}) = \max(\text{height}) = 3$



32

## Summary

33

## Key Terms

**tree:** a connected undirected graph with no simple circuits

**forest:** an undirected graph with no simple circuits

**rooted tree:** a directed graph with a specified vertex, called the root, such that there is a unique path to every other vertex from this root

**subtree:** a subgraph of a tree that is also a tree

parent of v in a rooted tree: the vertex u such that  $(u,v)$  is an edge of the rooted tree

**child of a vertex v in a rooted tree:** any vertex with v as its parent

**sibling of a vertex v in a rooted tree:** a vertex with the same parent as v

**ancestor of a vertex v in a rooted tree:** any vertex on the path from the root to v

**descendant of a vertex v in a rooted tree:** any vertex that has v as an ancestor

34

## Key Terms

**internal vertex:** a vertex that has children

**leaf:** a vertex with no children

**level of a vertex:** the length of the path from the root to this vertex

**height of a tree:** the largest level of the vertices of a tree

**m-ary tree:** a tree with the property that every internal vertex has no more than m children

**full m-ary tree:** a tree with the property that every internal vertex has exactly m children

**binary tree:** an m-ary tree with m = 2 (each child may be designated as a left or a right child of its parent)

**ordered tree:** a tree in which the children of each internal vertex are linearly ordered

35

## Key Results

### RESULTS

- A graph is a tree if and only if there is a unique simple path between every pair of its vertices.
- A tree with n vertices has  $n-1$  edges.

36

- ❑ **Balanced binary trees**
- ❑ **Properties of Trees**
- ❑ **Full binary tree**
- ❑ **Complete binary tree**
- ❑ **Binary Search Trees (BST)**

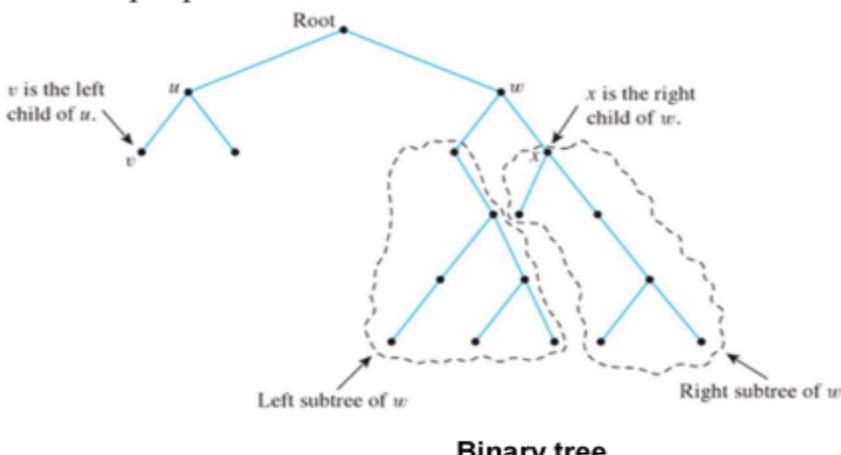
5

## Properties of Trees

Trees have several important properties other graphs do not have.

**Theorem:** The number of edges in a tree is one less than the number of its vertices:  $|E| = |V| - 1$   
i.e. A tree with  $n$  vertices has  $n-1$  edges.

**Theorem:** For every two vertices in a tree there always exists exactly one simple path from one of these vertices to the other.



6

## Balanced binary trees

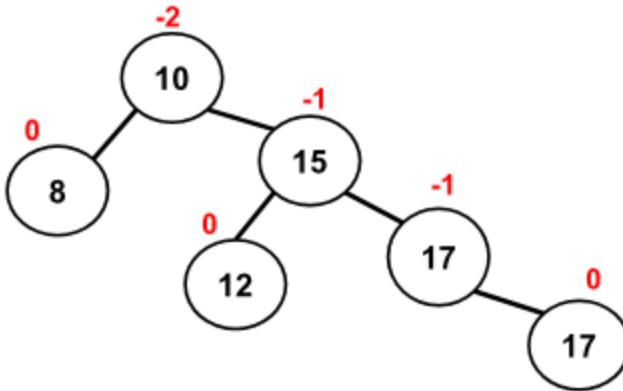
### Balance factor

**Definition :** Balance factor for any **vertex** is the difference between the height of the left subtree and right subtree of that node.     $bf = h_l - h_r$

$h_t$  = the height of a tree T is length of the longest path from root to leaf

## Example (9)

Find the balance factor for each vertex in the below tree.



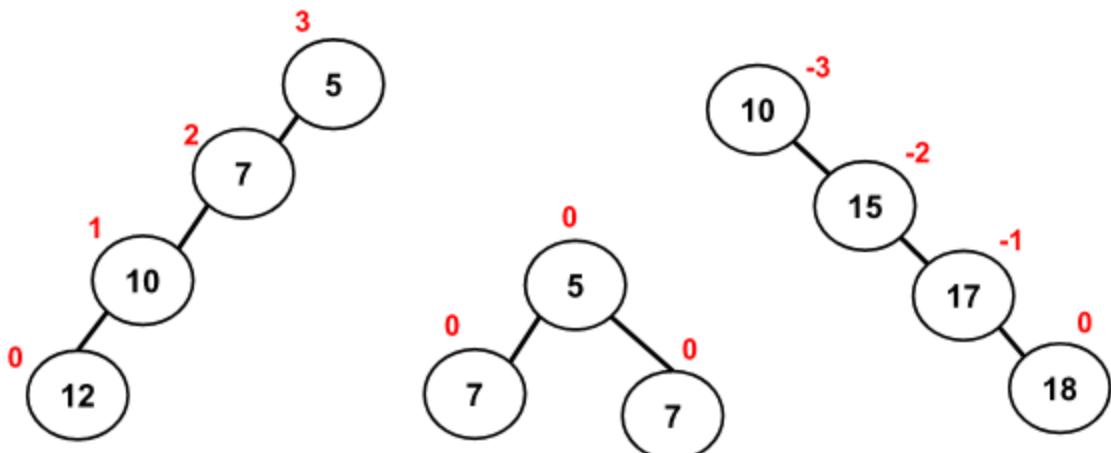
9

## Example (10)

Find the balance factor for each vertex in the below trees.

**The answer**

The balance factor for each vertex is  $bf = h_l - h_r$



10

# Balanced Binary trees

## Definition:

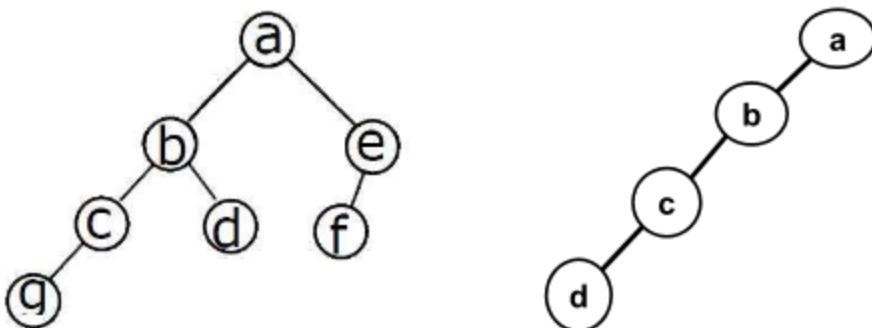
- A binary tree is balanced if the balance factor (bf) for every **vertex** is either 0 or +1 or -1.

**A height-balanced binary tree is:** a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

11

## Example (11)

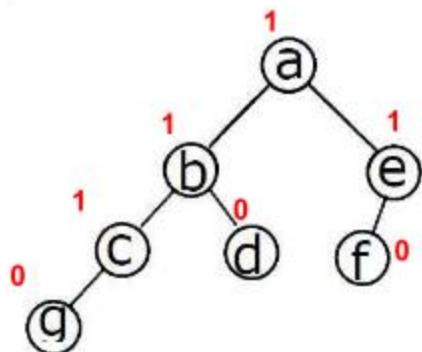
find the balance factor for each vertex in these figures.



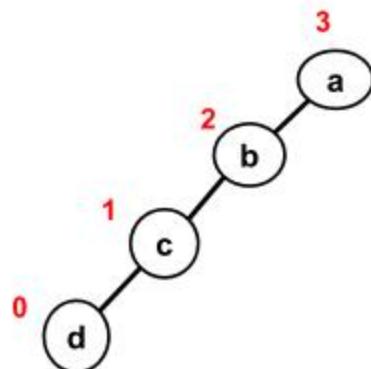
12

## Example (11)

find the balance factor for each vertex in these figures.



balanced binary tree

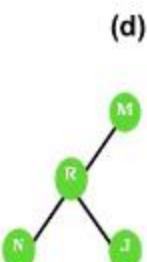
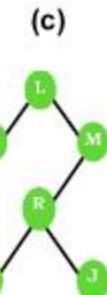
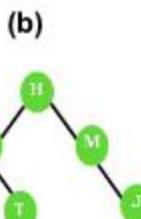
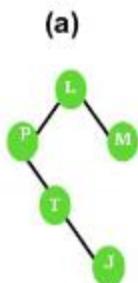


Unbalanced Binary Tree

13

## Example (12)

Which of following figures is a balanced binary tree?



Answer: b

Explanation: In Some tree diagrams, the root of tree has balance factor +2, so the tree is not balanced. If every node in the tree is balanced, then it's a balanced tree.

14

## Full binary Tree

## Full binary tree

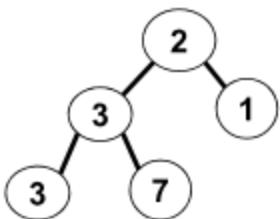
### Definition:

- A full binary tree is a tree in which each node **has exactly 0 or 2 children**. Both types of nodes can appear at all levels in the tree.

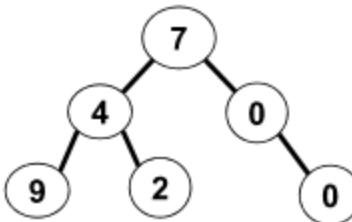
a binary tree T is full if each node is either a leaf or possesses exactly two child nodes.

## Example (13)

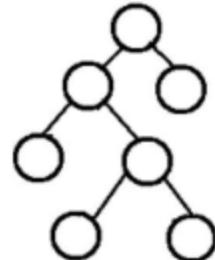
Which of following figures is a full binary tree?



Full tree



Not a full tree



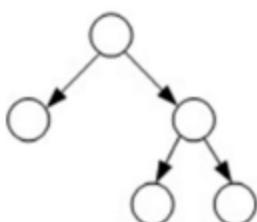
Full tree

17

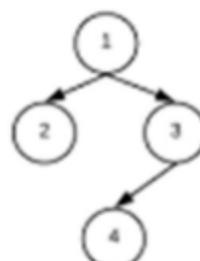
## Example (14)

Which of following figures is a full binary tree?

Full tree



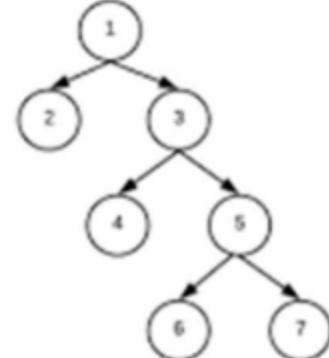
Not a full tree



Not full tree



Full tree

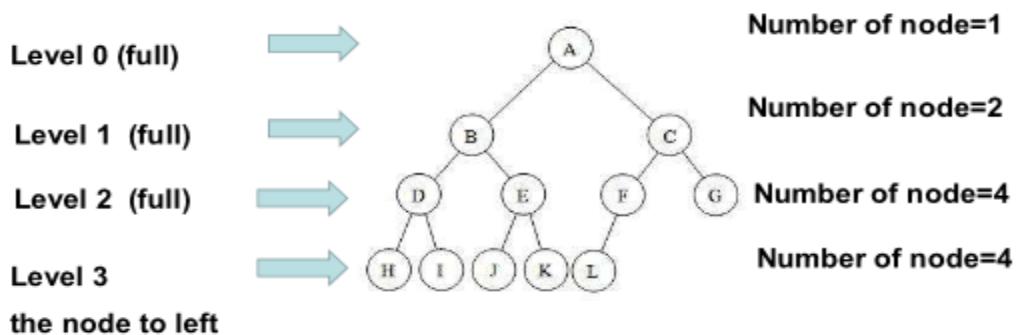


18

## complete binary tree

# Complete binary tree

A binary tree T with n levels is **complete** if all levels except possibly the last are **completely full**, and the last level has all its nodes to the **left** side.

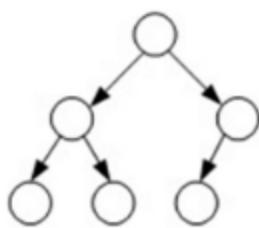


➤ Check level by level

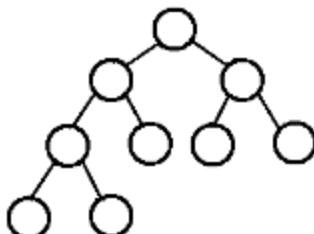
20

## Example (15)

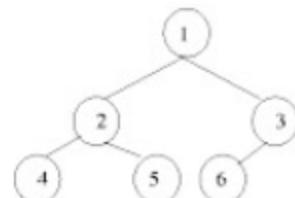
Which of following figures is a complete binary tree?



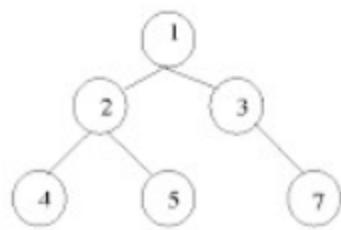
Complete tree



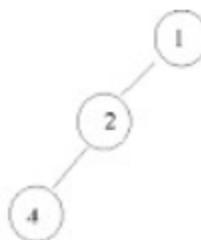
Complete tree



Complete tree



Not complete



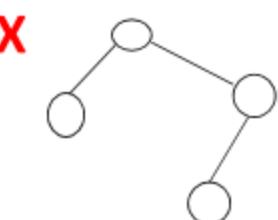
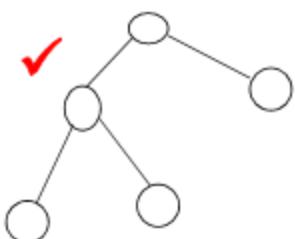
Not complete

➤ Check level by level

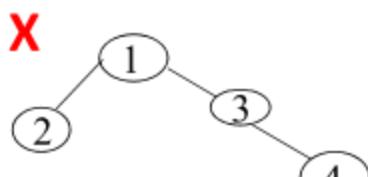
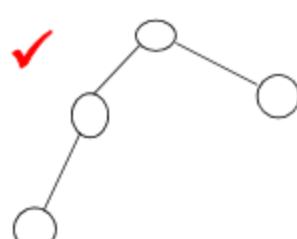
21

## Example (16)

Which of the following is a complete Binary Tree?



2  
2



➤ Check level by level

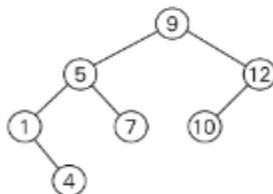
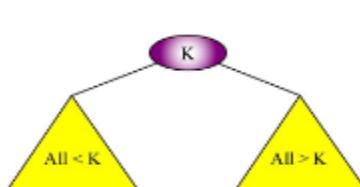
22

## A Binary Search Tree

23

## Binary Search Trees (BST)

- Searching for items in a list is one of the most important tasks that arises in computer science.
- Our primary goal is to implement a searching algorithm that finds items efficiently when the items are totally ordered.
- This can be accomplished through the use of a binary search tree,
- **Binary Search Trees (BST)**
  - A binary search tree (BST) is a binary tree with one extra property:
    - ✓ Vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.



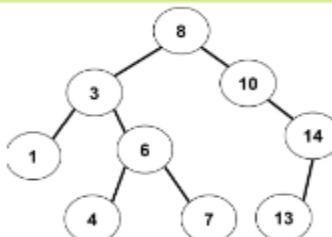
24

227

## A Binary Search Tree

**Binary Search Tree, is a node-based binary tree data structure which has the following properties:**

- The left subtree of a node contains only nodes with keys **lesser** than the node's key.
- The right subtree of a node contains only nodes with **keys** greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

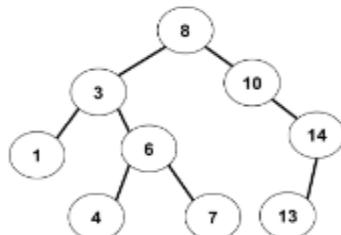


25

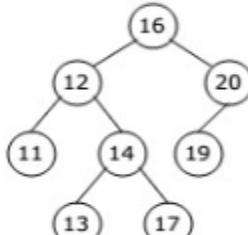
Binary Search Tree

## Example (17)

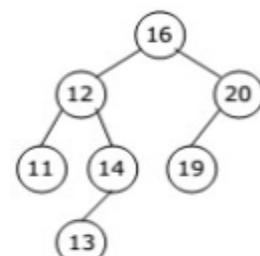
Which of these trees are BST?



Binary Search Tree



Not binary Search Tree

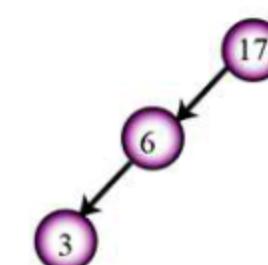


Binary Search Tree

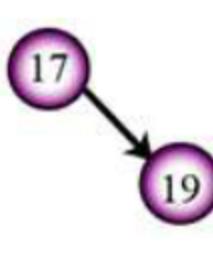
26

## Example (18)

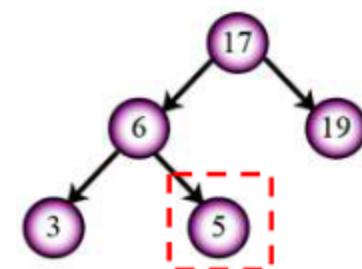
Which of these are binary search trees BSTs and which is not?



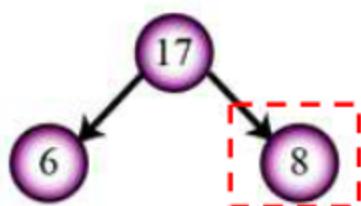
Binary Search Tree



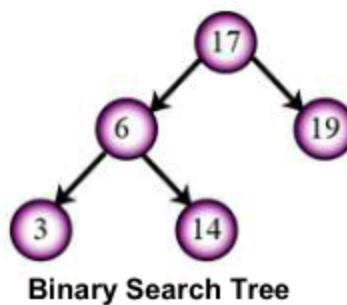
Binary Search Tree



Not Binary Search Tree



Not Binary Search Tree

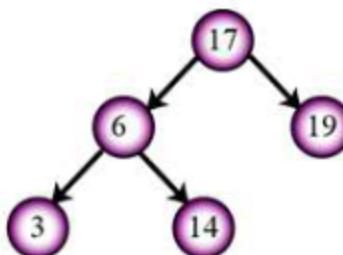


Binary Search Tree

27

## Search for element in a binary search tree (BST)

1. Start from root and compare the ITEM with root.
2. If the value is **below** root, we can say for sure that the value is not in the right subtree; **we need to only search in the left subtree**.
3. Otherwise, if the value is **above** root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.



28

## Example (19)

Search for element in a binary search tree (BST)

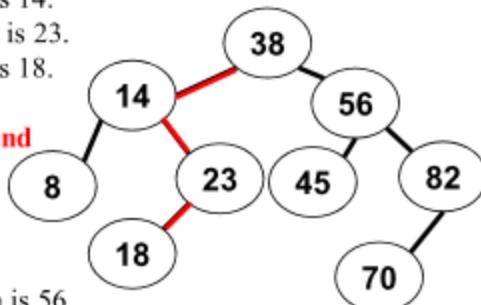
Check if the ITEMS=20, 70 are present in the next BST T or not, how many comparison you need to find them?

### Solution

Search for 20

1. Item=20<38, proceed to left child of 38, which is 14.
2. Item=20>14, proceed to right child of 14, which is 23.
3. Item=20<23, proceed to left child of 23, which is 18.
4. Item=20>18, and 18 does not have a right child,

After 4 comparisons we can say that 20 is not found



Search for 70

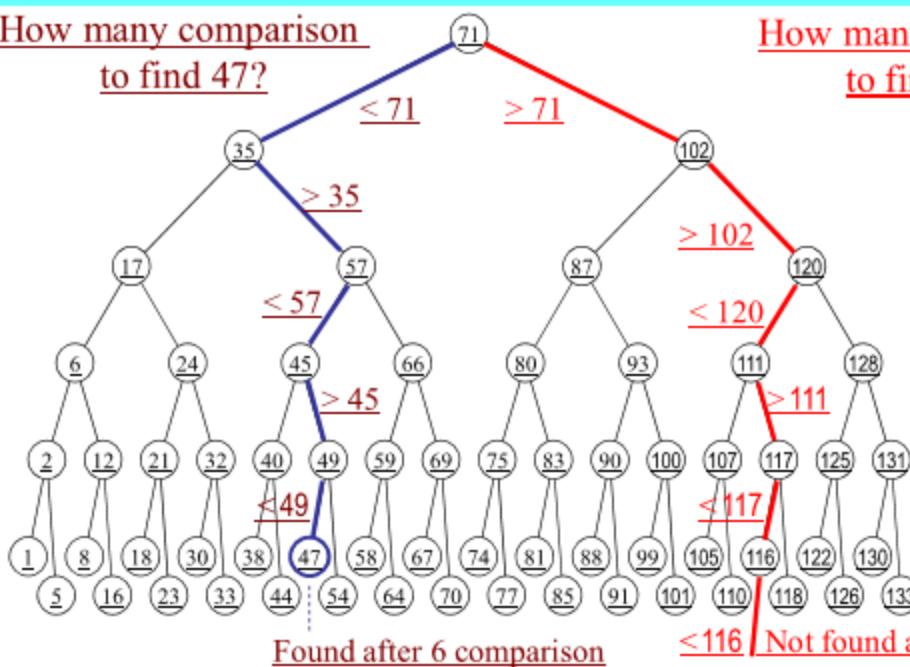
1. Item=70>38, proceed to right child of 38, which is 56.
2. Item=70>56, proceed to right child of 56, which is 82.
3. Item=70<82, proceed to left child of 82, which is 70.
4. Item=70=70, which is the item

The item found after 4 comparisons

29

## Example (20)

How many comparison  
to find 47?



How many comparison  
to find 112?

# A Binary Search Tree

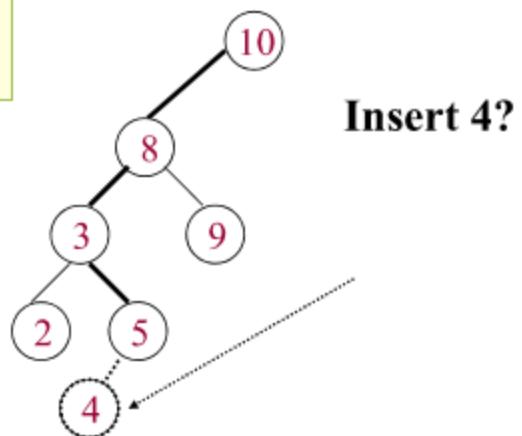
31

## Inserting a new key in a BST

### How to insert a new key?

The same procedure used for search also applies: Determine the location by searching. Search will fail. Insert new key where the search failed.

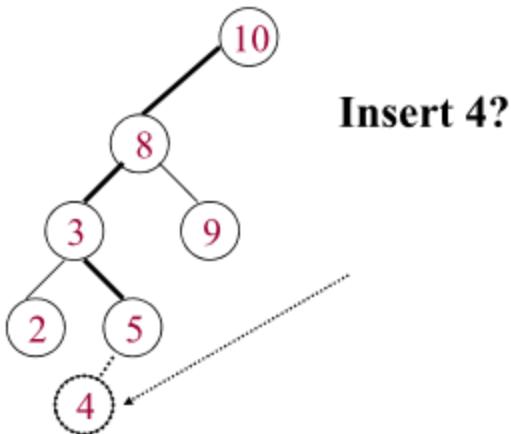
### Example (8):



32

## Example (21)

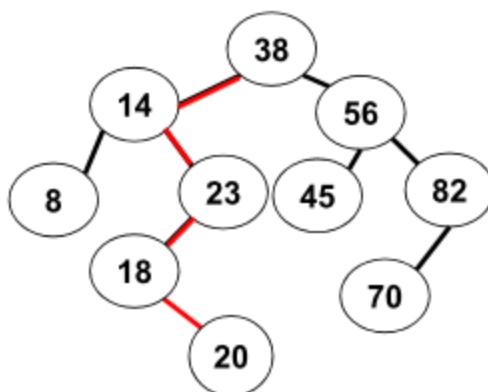
Inserting a new key in a BST



33

## Example (22)

Search for the ITEM=20 in the next BST T If it is not exist insert it in the appropriate position.



1. Item=20<38, proceed to left child of 38, which is 14.
2. Item=20>14, proceed to right child of 14, which is 23.
3. Item=20<23, proceed to left child of 23, which is 18.
4. Item=20>18, and 18 does not have a right child, insert 20 as the right child of 18 proceed to right child of 18, which is 20.

34

# Building a BST

Build the binary search tree of representing these numbers in the following order :

**45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48**

**A recursive procedure is used to build the binary search tree as follows:**

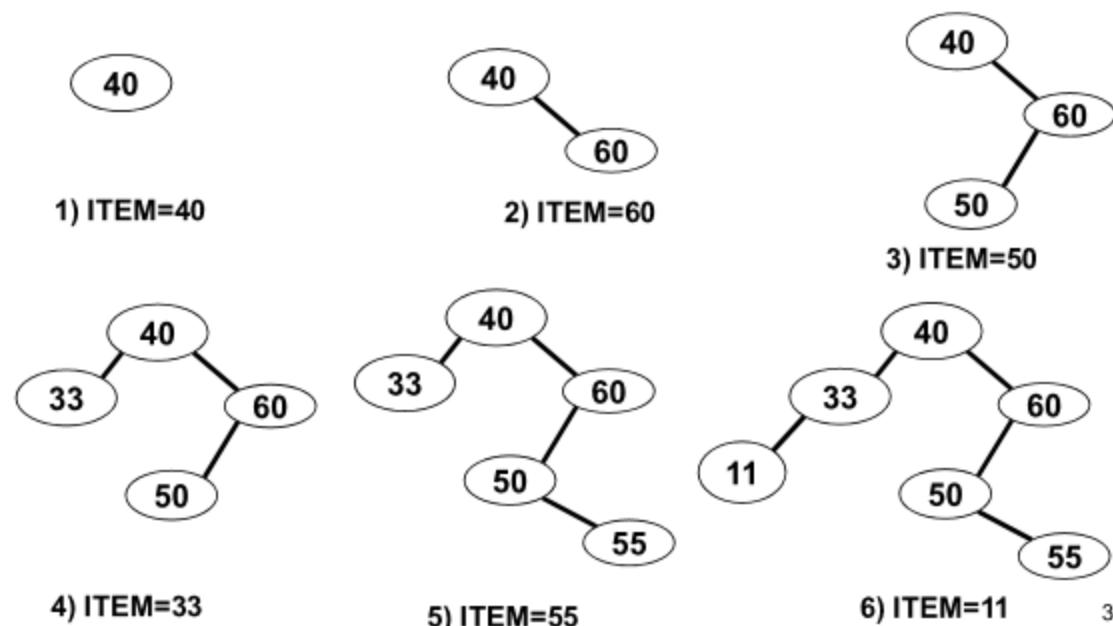
1. Start with a tree containing just **one** vertex, namely, the root. The first **item** in the list is assigned as the key of the **root**.
2. To add a new item, first compare it with the keys of vertices already in the tree,
  - ✓ starting at the root and
  - ✓ moving to the left if the item is **less than** the key of the respective vertex if this vertex has a left child, or
  - ✓ moving to the right if the item **is greater than** the key of the respective vertex if this vertex has a right child.
  - ✓ When the item **is less than** the respective vertex and this vertex **has no left child**, then a new vertex with this item as its key **is inserted as a new left child**.
  - ✓ Similarly, when the item **is greater than** the respective vertex and this vertex **has no right child**, then a new vertex with this item as its key is **inserted as a new right child**.

35

## Example (23)

Build a BST from these six numbers in order: **40, 60, 50, 33, 55, 11**

**Solution:** These numbers are inserted in order into an empty binary search tree as follows: these figures show the stages of building this tree.



36

## Example (24)

Build a BST from a sequence of nodes read one at a time

**Example:** Inserting **C A B L M** (in this order!)

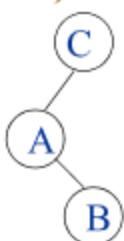
1) Insert C



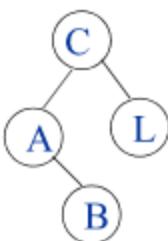
2) Insert A



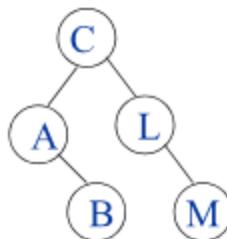
3) Insert B



4) Insert L



5) Insert M



37

## Example (25): Building a BST

Form a binary search tree BST for the following numbers

**45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48**

38

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

1) Insert 45



39

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

2) Insert 36

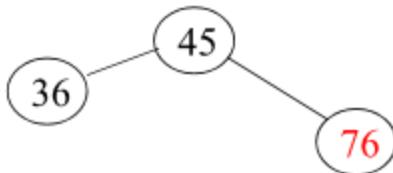


40

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

3) Insert 76

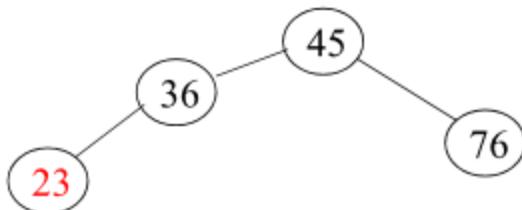


41

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

4) Insert 23

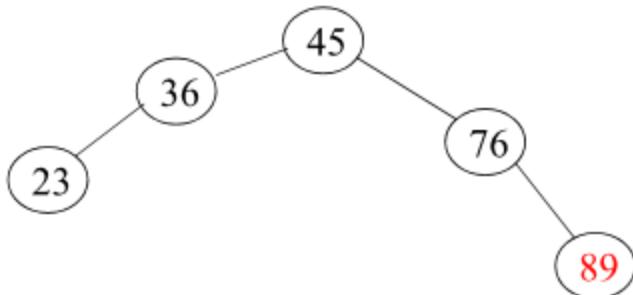


42

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

5) Insert 89

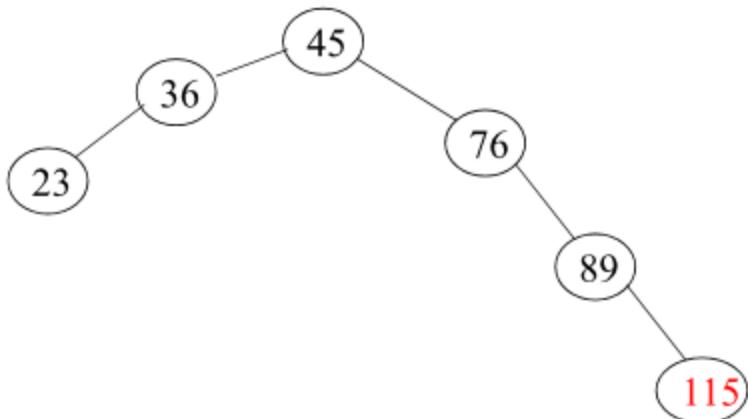


43

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

6) Insert 115

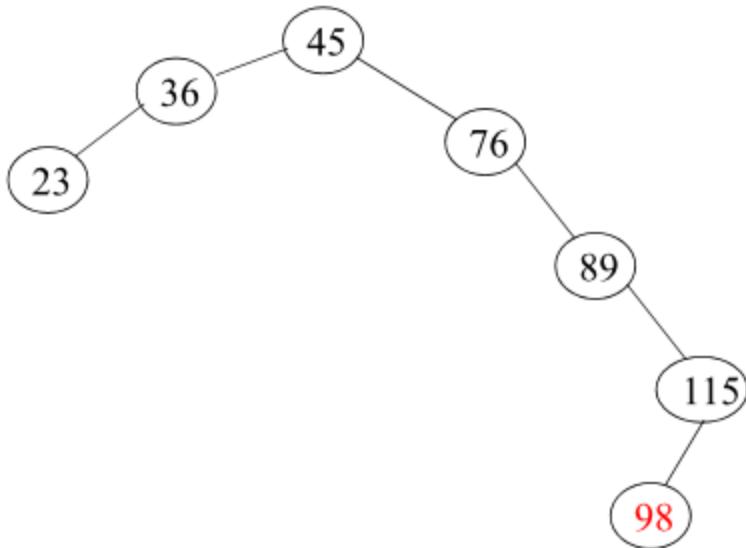


44

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

7) Insert 98

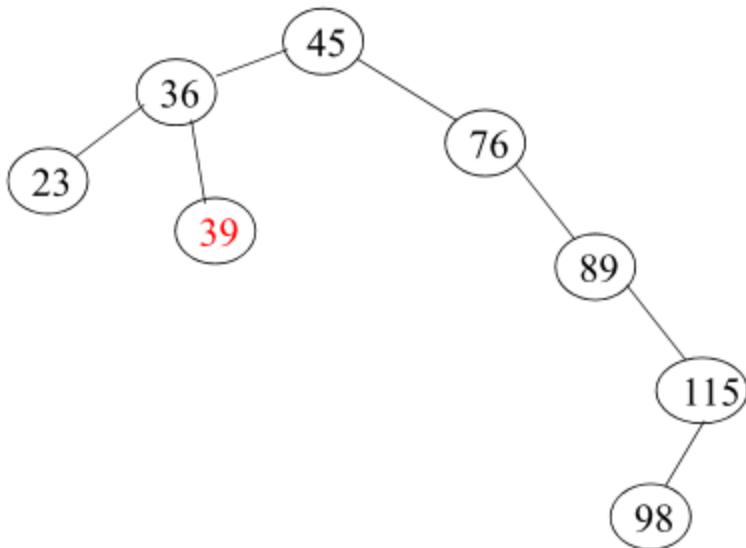


45

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

8) Insert 39

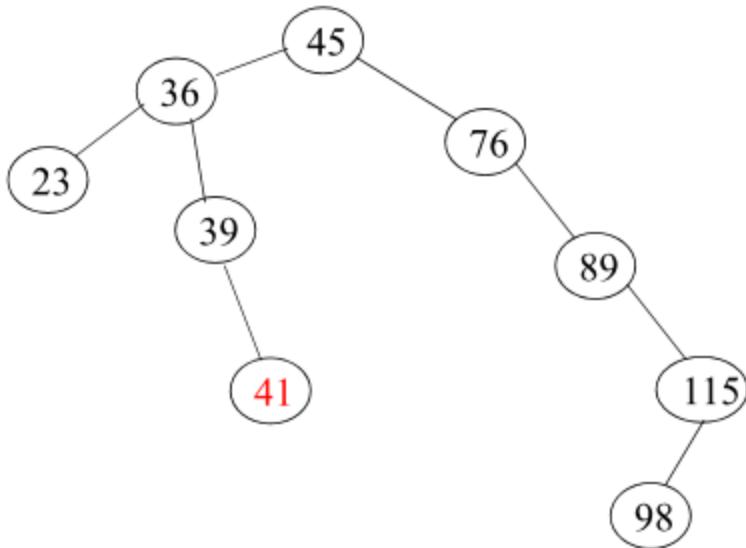


46

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

9) Insert 41

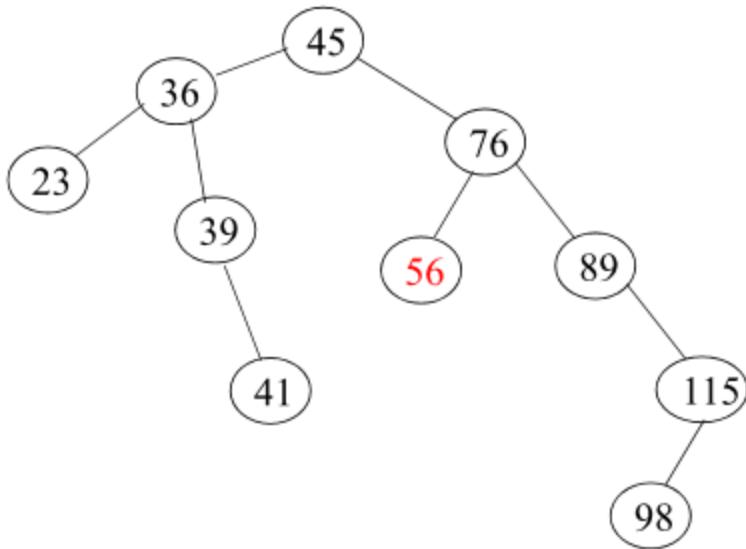


47

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

10) Insert 56

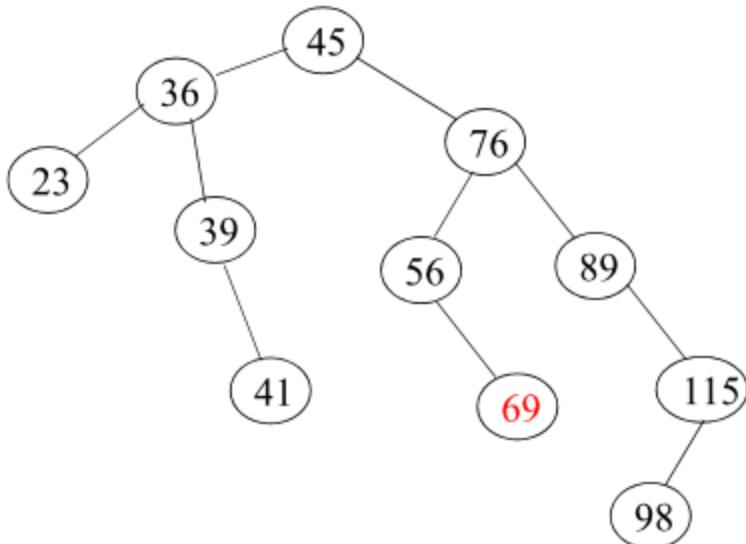


48

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, **69**, 48

11) Insert 69

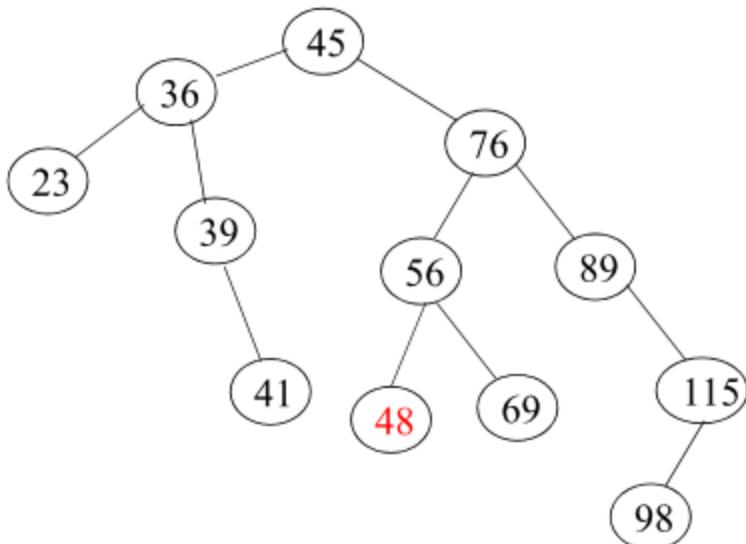


49

## Example (25): Building a BST

- 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, **69**, **48**

12) Insert 48



50

## Lecture: Overview

- Spanning Trees
- Minimum Spanning Trees (MST)
- Kruskal's Algorithm

5

## Spanning Trees

6

# Spanning Trees

## DEFINITION

In an undirected and connected simple graph  $G=(V, E)$ , a spanning tree is a subgraph that is a tree which includes all of the vertices of  $G$ , with minimum possible number of edges.

A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices.

## THEOREM :

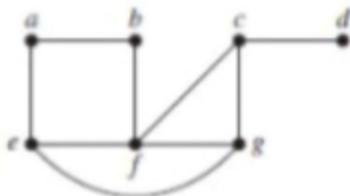
A simple graph is connected if and only if it has a spanning tree.

- A graph may have several spanning trees.

7

## Example (13)

Find a spanning tree of the simple graph  $G$  shown in this Figure.



The simple graph  $G$

### Solution

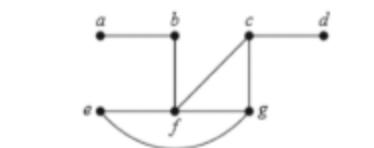
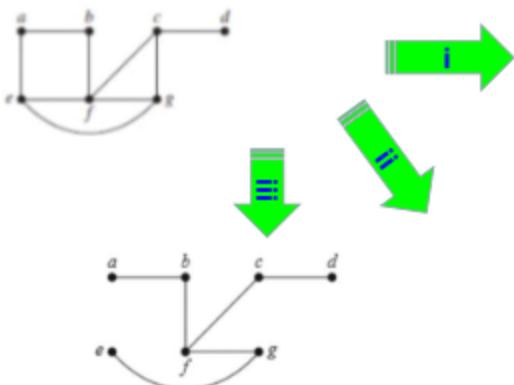
The graph  $G$  is connected, but it is not a tree because it contains simple circuits.

Producing a spanning tree for  $G$  can be done by removing edges that form simple circuits.

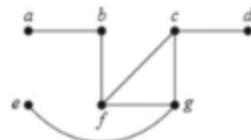
The sequence of edge removals used to produce the spanning tree is as follows:

8

## Example (13): Solution (I)



i. Edge  $\{a, e\}$  is removed



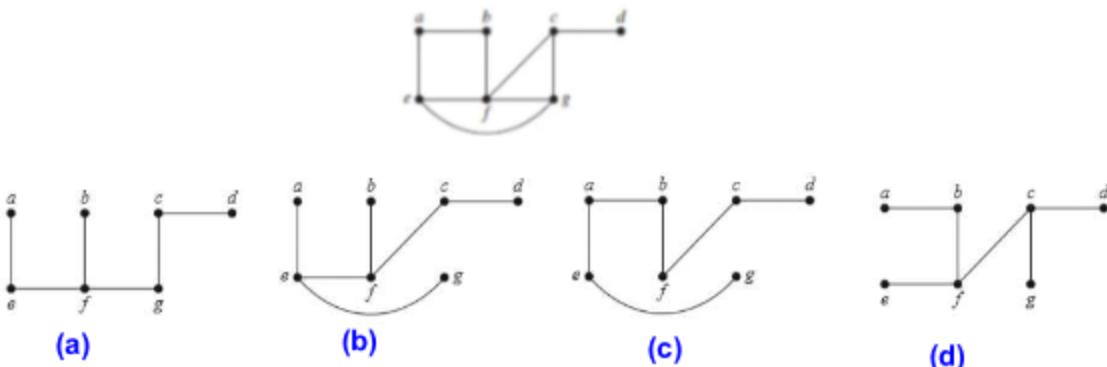
ii. Edge  $\{e, f\}$  is removed

- Remove the edge  $\{a, e\}$ . This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of  $G$ .
- Next remove the edge  $\{e, f\}$  to eliminate a second simple circuit.
- Finally, remove edge  $\{c, g\}$  to produce a simple graph with no simple circuits.

This subgraph is a spanning tree, because it is  
a tree that contains every vertex of  $G$ .

9

## Example (13): Solution (II)

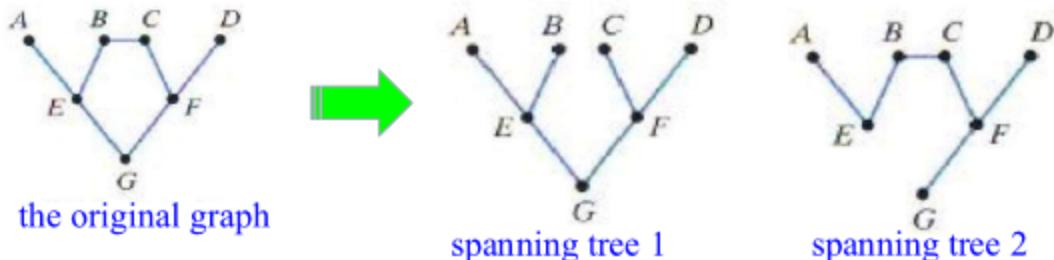


Spanning Trees of  $G$ .

The tree shown in the previous solution is not the only spanning tree of  $G$ . For instance, each of the trees shown in the above Figure is a four different spanning trees of  $G$ .

## Example (14)

Find a spanning tree of the simple graph G shown in this Figure.



### The answer

The two subgraphs in the above Figure, are spanning trees for the original graph. It is possible to start with a connected graph, retain all of its vertices, and remove edges until a spanning tree remains. Being a tree, the spanning tree must have one less edge than it has vertices.

Are there any other spanning trees?

11

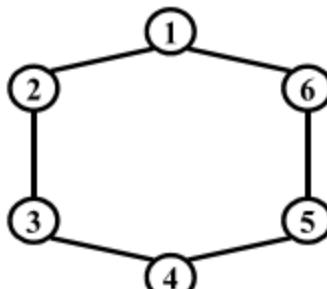
## Example (15): Spanning Tree

How many different Spanning tree ?

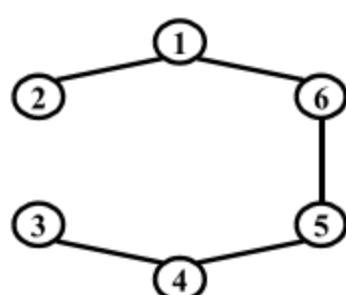
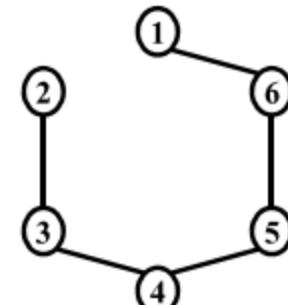
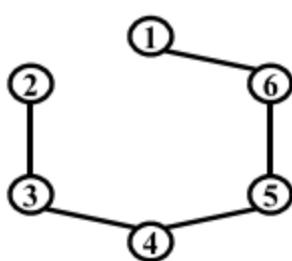
$$G = (V, E)$$

$$V = \{1, 2, 3, 4, 5, 6\} \quad |V| = 6$$

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



The answer



Spanning tree is a subgraph of a graph without cycle

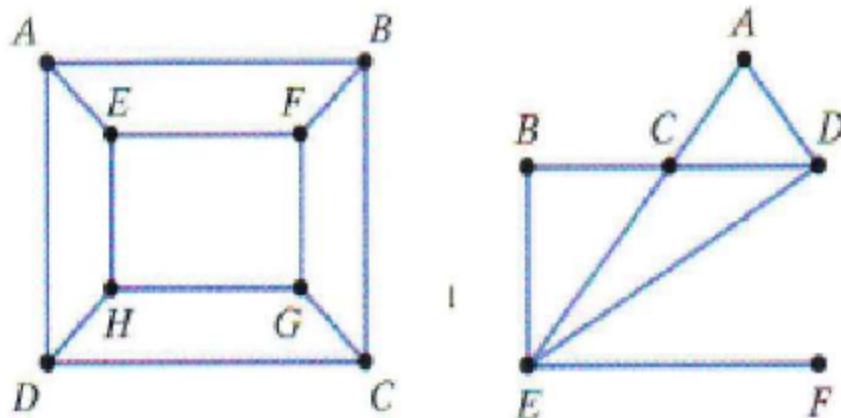
$$S \subseteq G \quad S = (V', E')$$

$$V' = V, \quad |E'| = |V| - 1$$

12

# Exercise (1)

Find a spanning tree for each of the graphs given below?



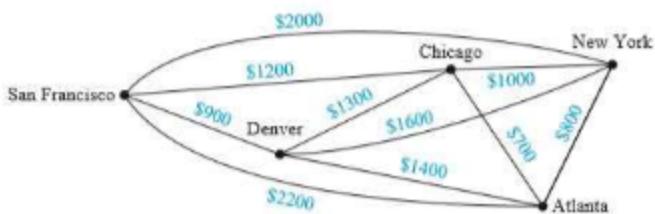
13

## Minimum Spanning Trees

14

# Minimum spanning tree

A company plans to build a communications network connecting its five computer centers. Any pair of these centers can be linked with a leased (الإيجار) telephone line. Which links should be made to ensure that there is a path between any two computer centers so that the total cost of the network is minimized? We can model this problem using the weighted graph shown in below Figure, where vertices represent computer centers, edges represent possible leased lines, and the weights on edges are the monthly lease rates of the lines represented by the edges. We can solve this problem by finding a spanning tree so that the sum of the weights of the edges of the tree is minimized. Such a spanning tree is called a minimum spanning tree.



A Weighted Graph Showing Monthly Lease Costs for Lines in a Computer Network

# Minimum spanning tree

## Algorithms for Minimum Spanning Trees

A wide variety of problems are solved by finding a spanning tree in a **weighted graph** such that the sum of the weights of the edges in the tree is a minimum.

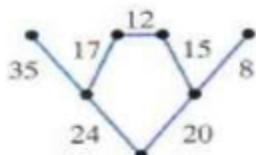
## DEFINITION

A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

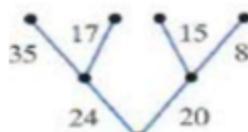
- That is, it is a spanning tree whose sum of edge weights is as small as possible.
- Number of edges in MST:  $V-1$  ( $V$  – no of vertices in Graph).
- The cost of the spanning tree is the sum of the weights of all the edges in the tree.

# Minimum Spanning Tree

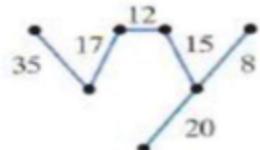
Many applied problems involve creating the most efficient network for a weighted graph. The weights often model distance costs or time, which we want to minimize. We do this by finding a minimum spanning tree.



(a) Original weighted graph



(b) A spanning tree with weighted  
 $35+24+20+8+17+15 = 119$



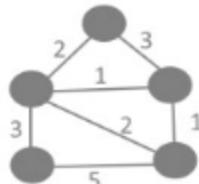
(c) A spanning tree with weighted  
 $35+17+12+15+20+8 = 107$

Figures (b) and (c) show two spanning trees for the weighted graph in Figure (a). The total weight for the spanning tree in Figure (c), 107, is less than that in Figure (b), 119.

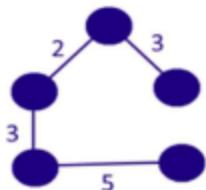
Is this the minimum spanning tree, or should we continue to explore other possible spanning trees whose total weight might be less than 107? 17

## Example (16):

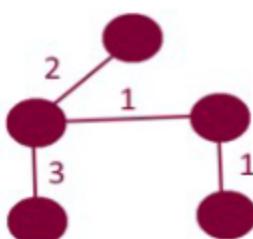
A very simple graph can have many spanning trees. Finding the minimum spanning tree by finding all possible spanning trees and comparing their weights would be too time-consuming. For example, How many different Spanning tree ?



The answer

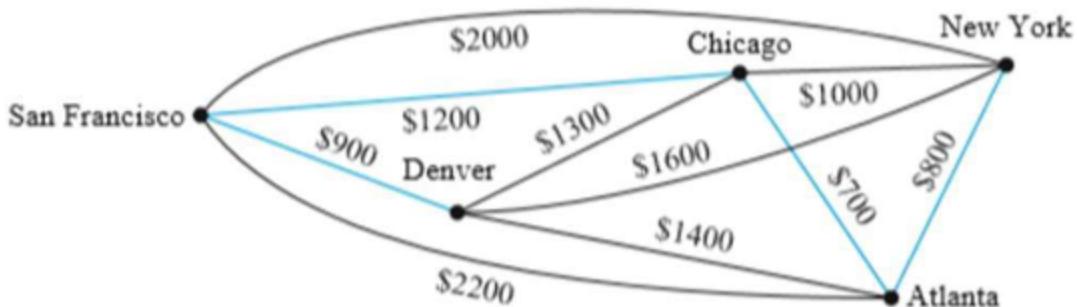


Spanning Tree#1  
Cost=13



Spanning Tree#2  
Cost=7, Minimum

## Example (17)



Choice	Edge	Cost
1	{Chicago, Atlanta}	\$700
2	{Atlanta, New York}	\$800
3	{Chicago, San Francisco}	\$1200
4	{San Francisco, Denver}	\$900
	Total	\$3600

FIGURE 2 A Minimum Spanning Tree for the Weighted Graph in Figure 1. <sup>19</sup>

## Kruskal's Algorithm

Is used for finding the Minimum Spanning Tree

# MST Using Kruskal's Algorithm

In 1956, the American mathematician Joseph Kruskal discovered a procedure that will always yield a minimum spanning tree for a weighted graph. The basic idea in Kruskal's Algorithm is to always pick the smallest available edge but avoid creating any circuits.

- Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph  $G = (V, E)$  as an **acyclic (without a cycle)** subgraph with  $|V| - 1$  edges for which the sum of the edge weights is the smallest.
- It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

21

## How this algorithm works?

**Below are the steps for finding MST using Kruskal's algorithm**

Number of edges in MST:  $V-1$  ( $V$  – no of vertices in Graph).

1. Sort the edges in ascending order of their weights.
2. Pick the edge with the smallest weight. Check if including this edge in spanning tree will form a cycle is Yes then ignore it if No then add it to spanning tree.
3. Repeat the step 2 till spanning tree has  $V-1$  ( $V$  – no of vertices in Graph).
4. Spanning tree with least weight will be formed, called Minimum Spanning Tree

22

# Kruskal's Algorithm Implementation

The implementation of Kruskal's Algorithm is explained in these steps:

➤ Step-01:

- Sort all the edges from low weight to high weight.

➤ Step-02:

- Take the edge with the lowest weight and use it to connect the vertices.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

➤ Step-03:

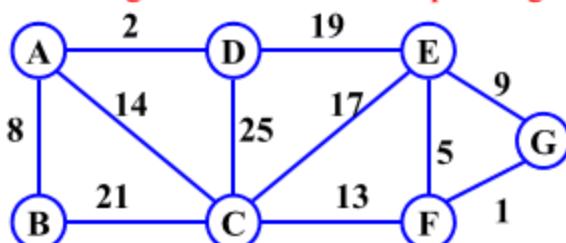
- Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

**Thumb Rule to Remember:** The above steps may be reduced to the following thumb rule

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

## Example (18)

Use Kruskal's Algorithm to find the minimum spanning tree for the above graph. Give the total weight of the minimum spanning tree.



After sorting:

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

The graph contains 7 vertices and 11 edges. So, the minimum spanning tree formed will be having  $(7 - 1) = 6$  edges.

So first the edges are sorted as in this table.

1. First, sort all the edges from low weight to high weight as in this table
2. Draw all the vertices on the paper.

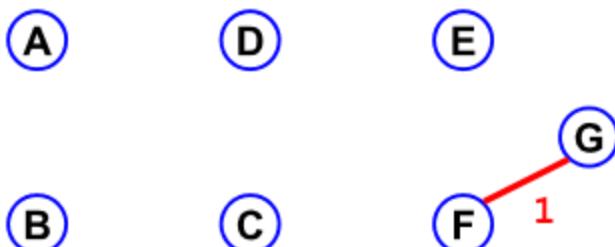


## Example (18)

Now pick all edges one by one from sorted list of edges

1. Pick edge FG: No cycle is formed, include it.

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

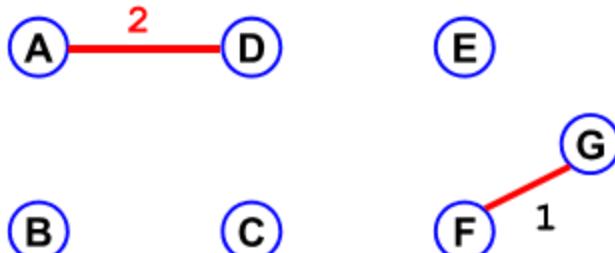


25

## Example (18)

2. Pick edge AD: No cycle is formed, include it.

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

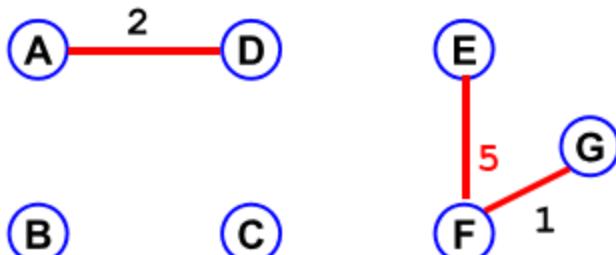


26

## Example (18)

3. Pick edge **EF**: No cycle is formed, include it.

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

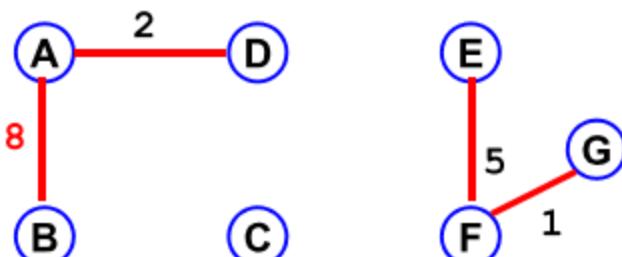


27

## Example (18)

4. Pick edge **AB**: No cycle is formed, include it.

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

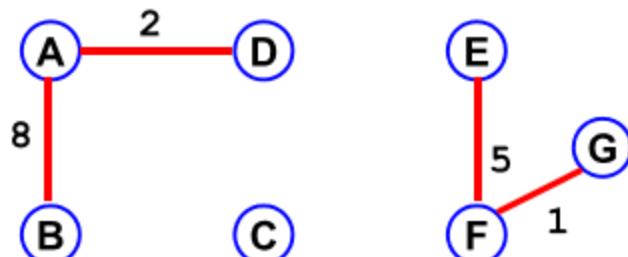


28

## Example (18)

5. Pick edge **EG**: Since including this edge results in cycle, discard it.

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

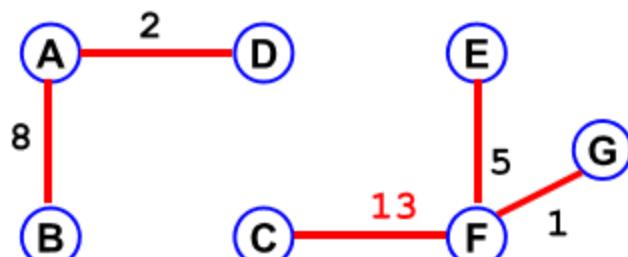


29

## Example (18)

6. Pick edge **CF**: No cycle is formed, include it.

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

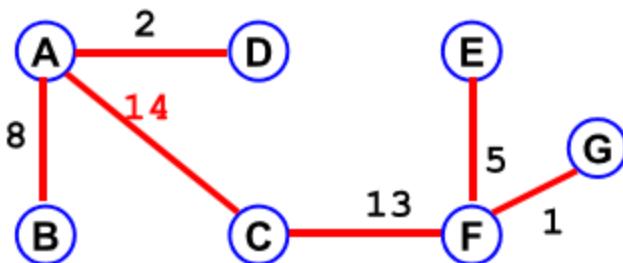


30

## Example (18)

7. Pick edge CA: No cycle is formed, include it.

W	Edge
1	FG
2	AD
5	EF
8	AB
9	EG
13	CF
14	CA
17	CE
19	DE
21	BC
25	CD

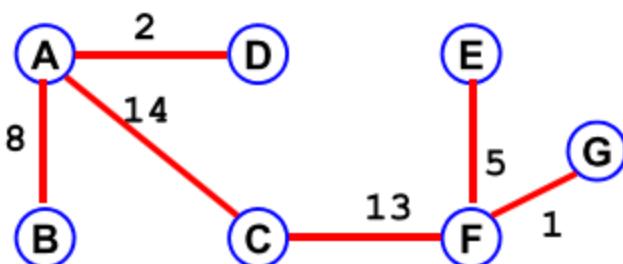


Since the number of edges included equals  $(V - 1)$ , the algorithm stops here.

31

## Example (18)

The minimum spanning tree



➤ The total weight of the MST

= Sum of all edge weights

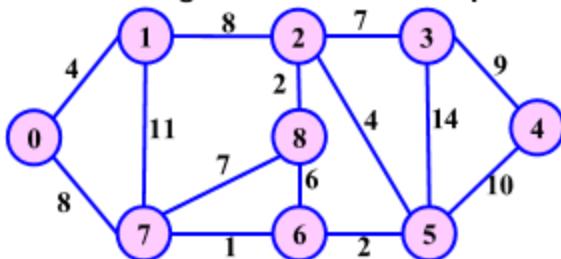
$$= 8 + 2 + 14 + 13 + 5 + 1$$

$$= 44 \text{ units}$$

32

## Example (19)

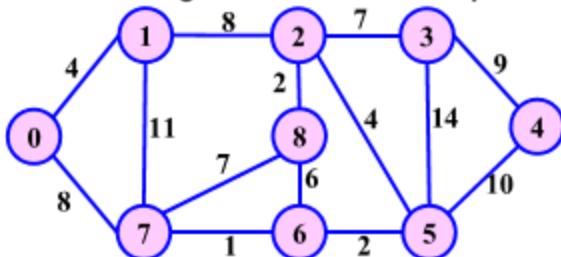
Use Kruskals' Algorithm to find the minimum spanning tree for the graph shown below. Give the total weight of the minimum spanning tree.



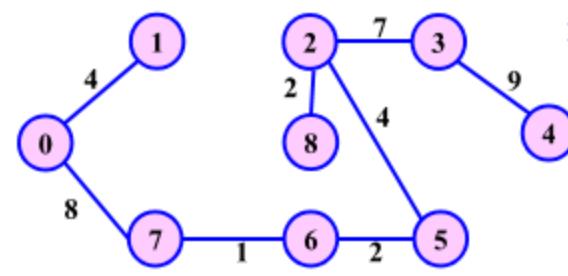
33

## Example (19)

Use Kruskals' Algorithm to find the minimum spanning tree for the graph shown below. Give the total weight of the minimum spanning tree.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having  $(9 - 1) = 8$  edges.



➤ weight of the MST  
= Sum of all edge weights  
 $4+8+1+2+4+2+7+9$   
 $= 39$  units

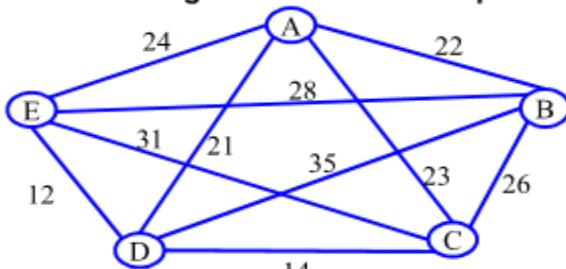
The minimum spanning tree

34

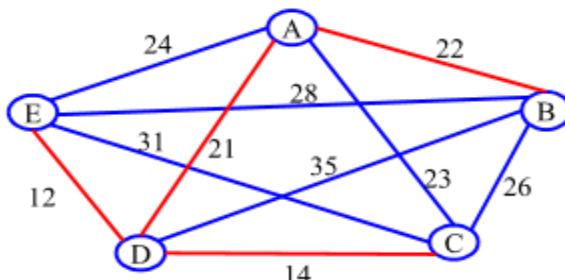
## Example (20)

Use Kruskals' Algorithm to find the minimum spanning tree for the graph shown below. Give the total weight of the minimum spanning tree.

Weight	Edge
12	ED
14	DC
21	AD
22	AB
23	AC
24	EA
26	BC
28	EB
31	EC
35	DB



The graph contains 5 vertices and 10 edges. So, the minimum spanning tree formed will be having  $(5 - 1) = 4$  edges.

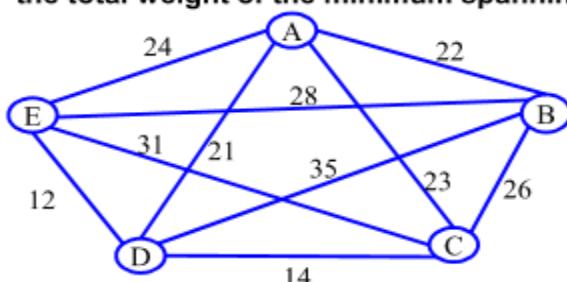


35

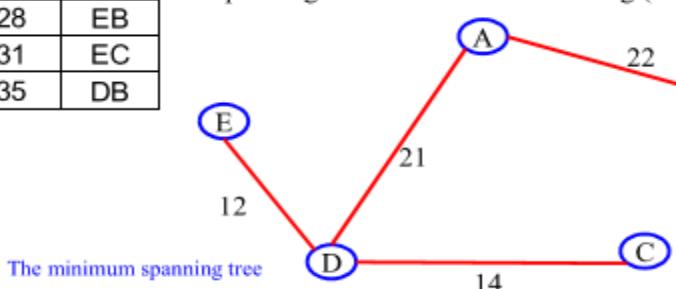
## Example (20)

Use Kruskals' Algorithm to find the minimum spanning tree for the graph shown below. Give the total weight of the minimum spanning tree.

Weight	Edge
12	ED
14	DC
21	AD
22	AB
23	AC
24	EA
26	BC
28	EB
31	EC
35	DB



The graph contains 5 vertices and 10 edges. So, the minimum spanning tree formed will be having  $(5 - 1) = 4$  edges.



➤ weight of the MST  
= Sum of all edge weights  
12+21+14+22  
= 69 units

The minimum spanning tree

36

# Applications

Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids.

Also used in:

- Approximating travelling salesman problem
- Approximating multi-terminal minimum cut problem
- Approximating minimum-cost weighted perfect Cluster Analysis
- Handwriting recognition
- Image segmentation
- Circuit design

37

# Summary and Key Terms

**binary search tree:** a binary tree in which the vertices are labeled with items so that a label of a vertex is greater than the labels of all vertices in the left subtree of this vertex and is less than the labels of all vertices in the right subtree of this vertex

**spanning tree:** a tree containing all vertices of a graph

**minimum spanning tree:** a spanning tree with smallest possible sum of weights of its edges

## RESULTS

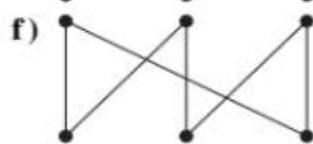
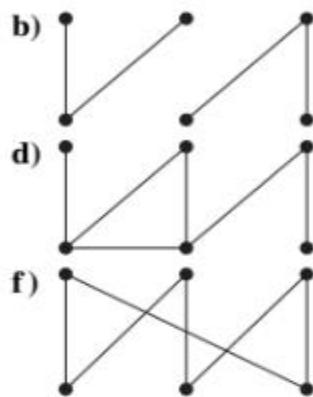
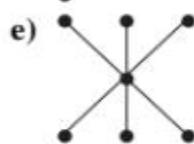
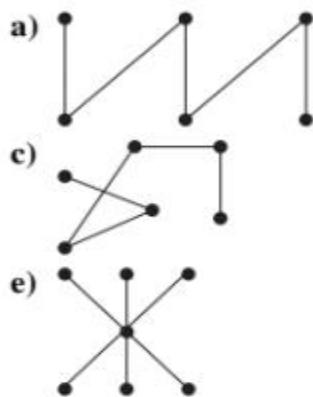
- Kruskal's algorithm: a procedure for producing a minimum spanning tree in a weighted graph that successively adds edges of least weight that are not already in the tree such that no edge produces a simple circuit when it is added

38

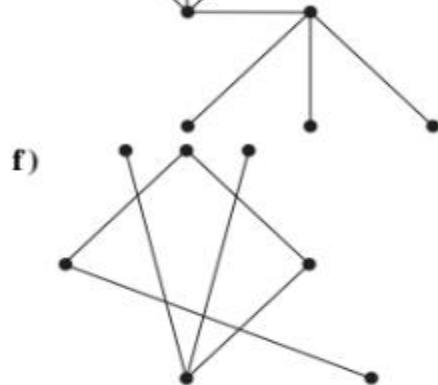
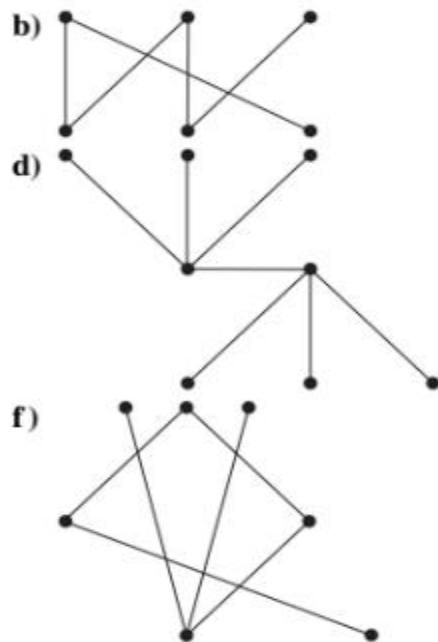
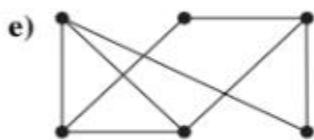
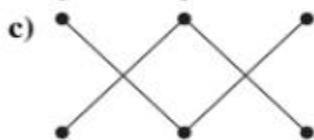
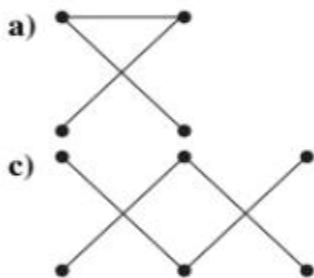
**Section No. 04**

**Answer the following Questions.**

1. Which of these graphs are trees?

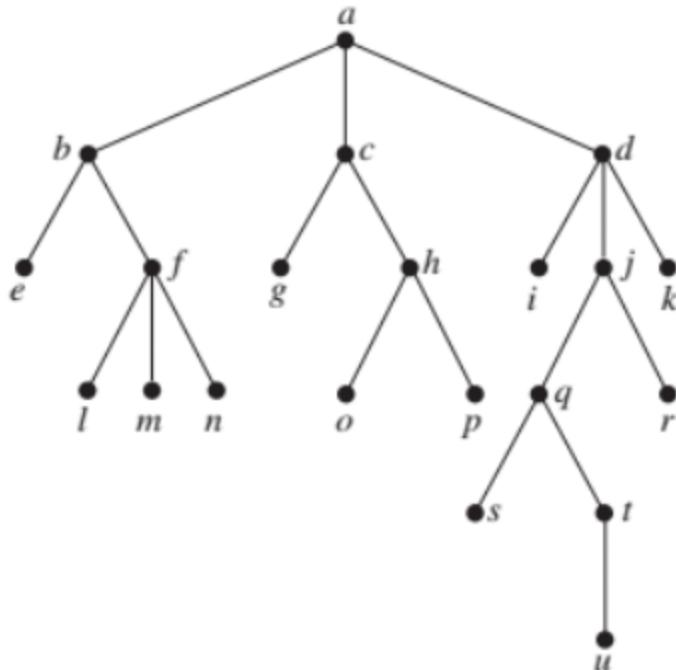


2. Which of these graphs are trees?



3. Answer these questions about the rooted tree illustrated.

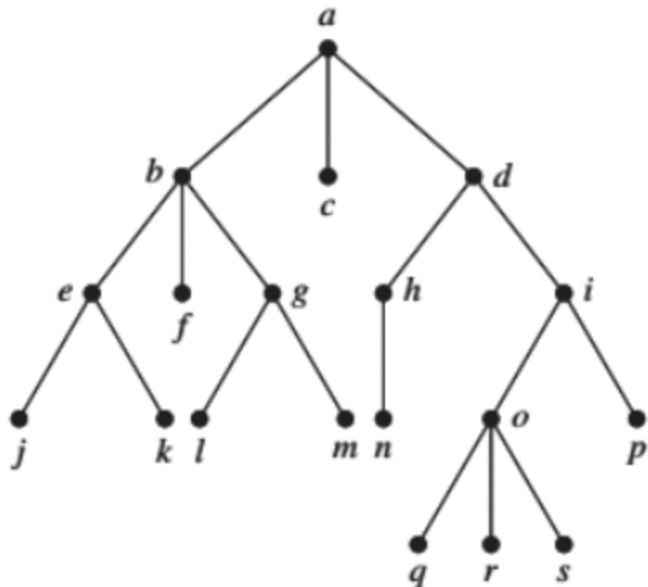
- a) Which vertex is the root?
- b) Which vertices are internal?
- c) Which vertices are leaves?
- d) Which vertices are children of  $j$ ?
- e) Which vertex is the parent of  $h$ ?
- f) Which vertices are siblings of  $o$ ?
- g) Which vertices are ancestors of  $m$ ?
- h) Which vertices are descendants of  $b$ ?



4. Is the rooted tree in Exercise 3 a full  $m$ -ary tree for some positive integer  $m$ ?
5. What is the level of each vertex of the rooted tree in Exercise 3?
6. Draw the subtree of the tree in Exercise 3 that is rooted at
  - a)  $a$ .
  - b)  $c$ .
  - c)  $e$ .
7. How many edges does a tree with 10,000 vertices have?
8. Construct a complete binary tree of height 4 and a complete 3-ary tree of height 3.
9. How many vertices and how many leaves does a complete  $m$ -ary tree of height  $h$  have?
10. How many vertices does a full 5-ary tree with 100 internal vertices have?
11. How many edges does a full binary tree with 1000 internal vertices have?
12. How many leaves does a full 3-ary tree with 100 vertices have?

13. Answer these questions about the rooted tree illustrated.

- a) Which vertex is the root?
- b) Which vertices are internal?
- c) Which vertices are leaves?
- d) Which vertices are children of  $j$ ?
- e) Which vertex is the parent of  $h$ ?
- f) Which vertices are siblings of  $o$ ?
- g) Which vertices are ancestors of  $m$ ?
- h) Which vertices are descendants of  $b$ ?



## Section No. 05

### Answer the following Questions.

1. Build a binary search tree for the words **banana, peach, apple, pear, coconut, mango, and papaya** using alphabetical order.
2. Build a binary search tree for the words **oenology, phrenology, campanology, ornithology, ichthyology, limnology, alchemy, and astrology** using alphabetical order.
3. How many comparisons are needed to locate or to add each of these words in the search tree for Exercise 1, starting fresh each time?

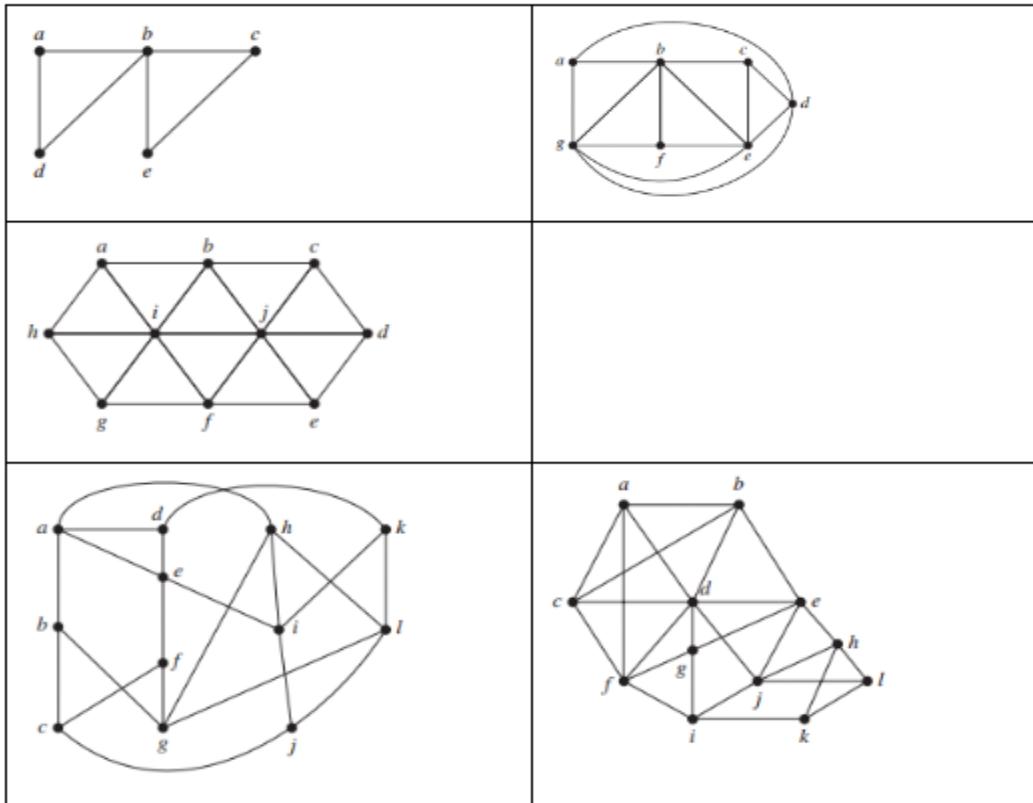
Pear	banana
kumquat	orange

4. How many comparisons are needed to locate or to add each of the words in the search tree for Exercise 2, starting fresh each time?

palmistry	etymology
paleontology	glaciology

5. Form a binary search tree for the words **mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry** (using alphabetical order).
6. Form a binary search tree BST for the following numbers **45, 10, 7, 90, 12, 50, 13, 39, 57**

7. find a spanning tree for the graph shown by removing edges in simple circuits.



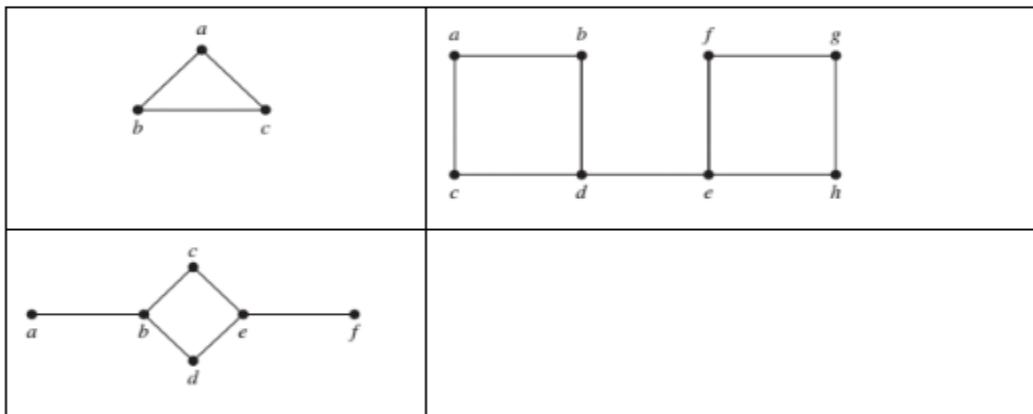
8. Find a spanning tree for each of these graphs.

a)  $K_5$   
d)  $Q_3$

b)  $K_{4,4}$   
e)  $C_5$

c)  $K_{1,6}$   
f)  $W_5$

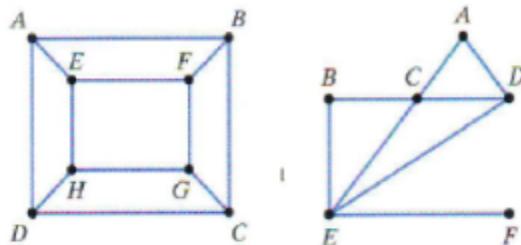
9. draw all the spanning trees of the given simple graphs.



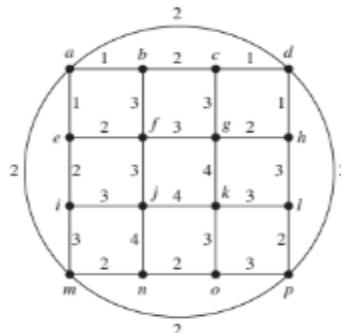
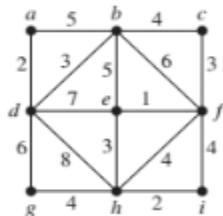
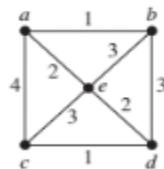
10. How many different spanning trees does each of these simple graphs have?

- a)  $K_3$       b)  $K_4$       c)  $K_{2,2}$       d)  $C_5$

11. Find a spanning tree for each of the graphs given below?



12. Use Kruskals' Algorithm to find a minimum spanning tree for the given weighted graph. Give the total weight of the minimum spanning tree.



# **Chapter (6)**

## **Algorithms, the Growth of Functions**

## Algorithms

- An **algorithm** is a finite set of precise instructions for performing a computation or for solving a problem.
- The **time** required to solve a problem depends on the number of **steps** it uses.
- **Growth functions** are used to estimate the number of **steps** an algorithm **uses** as its input **grows**.

7

## Analysis of Algorithms

### Goal:

- To analyze and compare algorithms in terms of *running time* and *memory requirements* (i.e. **time** and **space complexity**)

**Time efficiency:** how fast an algorithm runs.

**Space efficiency:** the space an algorithm requires.

- Typically, algorithms **run longer** as the size of its **input increases**
- We are interested in how **efficiency** scales w.r.t **input size**.

8

## Measuring Input Sizes

- Almost all algorithms run longer on larger inputs. Therefore, it is logical to investigate an algorithm's efficiency as a **function of some parameter  $n$**   $f(n)$  indicating the algorithm's input size.
- Input size depends on the problem.
  - **Example 1:** what is the input size of the problem of sorting  $n$  numbers?
  - **Example 2:** what is the input size of adding two  $n$  by  $n$  matrices?

9

## Measuring Input Sizes

**Input size** (number of elements in the input)

- ✓ size of an *array* or a *matrix*
- ✓ # of bits in the binary representation of the input
- ✓ # vertices and/or # edges in a graph, etc.

Problem	Input size measure
Searching for key in a list of $n$ items	Number of list's items, i.e. $n$
Multiplication of two matrices	Matrix dimensions or total number of elements
Checking primality of a given integer $n$	$n$ 'size = number of digits (in binary representation)
Typical graph problem	#vertices and/or edges

10

## Units for Measuring Running Time

- Measure the **execution time**?
- Should we measure **the running time** using standard unit of time **measurements**, such as **seconds, minutes**?
- Not a good idea! **because:**
  - It varies for different microprocessors!
  - Depends on the speed of the computer.
  - Depends on the quality of a program implementing the algorithm

11

## Units for Measuring Running Time

- Count the **number of statements executed**?

**Yes, but you need to be very careful!**
- One possible approach is to count the number of times each of the **algorithm's operations** is executed.
  - Difficult and usually unnecessary
- Associate a "cost" with each statement.
- Find the "total cost" by multiplying the cost with the total number of times each statement is executed.

12

## Example (1)

Algorithm X	Cost
sum = 0;	$c_1$
for(i=0; i<N; i++)	$c_2$
for(j=0; j<N; j++)	$c_3$
sum += arrY[i][j];	$c_4$
-----	

$$\text{Total Cost} = c_1 + c_2 * (N+1) + c_3 * N * (N+1) + c_4 * N^2$$

13

## Units for Measuring Running Time

- Count the number of times an algorithm's **basic operation** is executed.
- **Basic operation:** the operation that contributes the most to the total running time.
- For example, the basic operation is usually the **most time-consuming operation** in the algorithm's innermost loop.

14

## Basic operation examples

Problem	Basic operation
Searching for key in a list of $n$ items	Key comparison
Multiplication of two matrices	Multiplication of two numbers
Checking primality of a given integer $n$	Division
Typical graph problem	Visiting a vertex or traversing an edge

15

## Theoretical Analysis of Time Efficiency

- Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size.

$$T(n) \approx c_{op} C(n)$$

Diagram illustrating the components of time efficiency analysis:

- input size
- running time
- execution time for the basic operation
- Number of times the basic operation is executed

Red arrows point from each component to its corresponding term in the formula  $T(n) \approx c_{op} C(n)$ .

The efficiency analysis framework ignores the multiplicative constants of  $C(n)$  and focuses on the orders of growth of the  $C(n)$ .

16

## Theoretical Analysis of Time Efficiency

Assuming

$$C(n) = (1/2)n(n-1) \approx (\frac{1}{2}) n^2 \approx n^2$$

**How much longer will the algorithm run if we double the input size?**

The answer is about four times longer.

- To describe running time of algorithms, we consider the size of input is very large. So, we can ignore constants in describing running time.

### Types of Analysis

## Types of Analysis : Worst case Efficiency

### Worst case Efficiency

- The maximum value of  $f(n)$  for any possible input.
- Provides an upper bound on running time
- Efficiency (# of times the basic operation will be executed) for the worst case input of size  $n$ .
- The algorithm runs the longest among all possible inputs of size  $n$ .
- For the input of size  $n$ , the running time is  $C_{\text{worst}}(n) = n$ .

The maximum value of  $f(n)$  for any possible input.

Worst Case- searching for a key at last index

Worst case time :  $W(n)=O(n)$

19

## Types of Analysis : Best-case efficiency

### Best-case efficiency

- The minimum possible value of  $f(n)$ 
  - Provides a lower bound on running time
  - Efficiency (# of times the basic operation will be executed) for the best case input of size  $n$ .
  - The algorithm runs the fastest among all possible inputs of size  $n$ .
  - In sequential search, If we search a first element in list of size  $n$ . (i.e. first element equal to a search key), then the running time is  $C_{\text{best}}(n) = 1$

The minimum possible value of  $f(n)$ ,

- Best Case- searching for element presented at the first index
- Best case time - is constant  $B(n)=O(1)$

20

## Types of Analysis : Average-case efficiency

### Average-case efficiency

- The expected value of  $f(n)$
- Provides a prediction about the running time
- Assumes that the input is random
- The average case efficiency lies between **best** case and **worst** case.
- Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.
- NOT the average of worst and best case
- How to find the average case efficiency?

**Lower Bound ≤ Running Time ≤ Upper Bound**

21

### Example (2)

- Linear search



A	8	6	12	5	9	7	4	3	16	18
index	0	1	2	3	4	5	6	7	8	9

Key=7 C=6 successful

Key=20 C=10 not successful

Key operation is comparison???

22

## Example (2): Best-Case Efficiency

- Linear search

A	8	6	12	5	9	7	4	3	16	18
	0	1	2	3	4	5	6	7	8	9

Best Case- searching for element presented at the first index

Best case time - is constant 1 → O(1)

$$B(n)=O(1)$$

23

## Example (2): Worst-Case Efficiency

- Linear search

A	8	6	12	5	9	7	4	3	16	18
	0	1	2	3	4	5	6	7	8	9

Worst Case- searching for a key at last index

Worst case time - n → O(n)

$$W(n)=O(n)$$

24

## Example (2): Average-Case Efficiency

- Linear search

A	8	6	12	5	9	7	4	3	16	18
	0	1	2	3	4	5	6	7	8	9

$$\text{Average Case} = \frac{\text{all possible case time}}{\text{number of cases}}$$

It is very difficult not possible for all algorithms  
Most of the time it will be close to the worst case

Average case time –

1-In case of the key is the first →

2-in case of the key is the second →

.....

$$\text{Average} = \frac{1+2+3+\dots+n}{n} = \frac{n(n+1)}{2n} = \frac{(n+1)}{2}$$

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

25

## Asymptotic Analysis

## Asymptotic Analysis

- To compare **two** algorithms with running times  $f(n)$  and  $g(n)$ , we need a **rough measure** that characterizes **how fast each function grows** with respect to **n**
- In other words, we are interested in how they behave **asymptotically** (i.e. for large  $n$ ) (called **rate of growth**)

27

## Asymptotic Notations And Its Properties

- Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program.
- The efficiency analysis framework concentrates on the **order of growth** of an algorithm's **basic operation** count as the principal indicator of the algorithm's efficiency.

28

## Asymptotic Notations And Its Properties

- To compare and rank such orders of growth, computer scientists use three notations, they are:
  - ❑ O-Big oh notation (Upper bound)
  - ❑  $\Omega$ -Big omega notation (Lower bound)
  - ❑  $\Theta$ -Big theta notation (Average bound)

29

## Asymptotic Analysis

- **Big O notation:** asymptotic “**less than**” or “at most”:

$$f(n)=O(g(n)) \text{ implies: } f(n) \leq g(n)$$

- $O(g(n))$  is the set of all functions with a **smaller or same** order of growth as  $g(n)$ .

30

## Asymptotic Analysis

➤  **$\Omega$  notation:** asymptotic “greater than” or “at least”:

$$f(n) = \Omega(g(n)) \text{ implies: } f(n) \geq g(n)$$

- $\Omega(g(n))$  stands for the set of all functions with **a larger or same** order of growth as  $g(n)$ .

31

## Asymptotic Analysis

➤  **$\Theta$  notation:** asymptotic “equality” or “exactly”:

$$f(n) = \Theta(g(n)) \text{ implies: } f(n) = g(n)$$

- $\Theta(g(n))$  is the set of all functions that have **the same order of growth** as  $g(n)$ .

32

## Big-O Notation

➤ We say

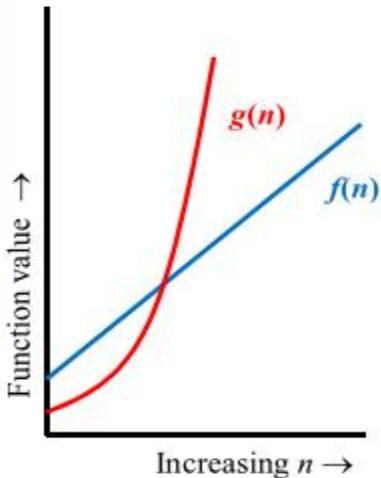
$f(n) = 7n+18$  is **order  $n$** , or  **$O(n)$**

It is, at most, roughly *proportional* to  $n$ .

$g(n) = 3n^2+5n+4$  is **order  $n^2$** , or  **$O(n^2)$** .

It is, at most, roughly *proportional* to  $n^2$ .

- In general, any  $O(n^2)$  function is faster-growing than any  $O(n)$  function.



## Example (3)

- $n^4 + 100n^2 + 10n + 50 \rightarrow O(n^4)$

$$10n^3 + 2n^2 \rightarrow O(n^3)$$

$$n^3 - n^2 \rightarrow O(n^3)$$

- constants

10 is  $O(1)$

1273 is  $O(1)$

35

## Example (4)

- what is the rate of growth for *Algorithm X* studied earlier (in Big O notation)?

<b>Algorithm X</b>	<b>Cost</b>
sum = 0;	$c_1$
for(i=0; i<N; i++)	$c_2$
for(j=0; j<N; j++)	$c_3$
sum += arrY[i][j];	$c_4$

$$\text{Total Time} = c_1 + c_2 * (N+1) + c_2 * N * (N+1) + c_3 * N^2$$

If  $c_1$ ,  $c_2$ ,  $c_3$ , and  $c_4$  are constants then  $\text{Total Time} = O(N^2)$

36

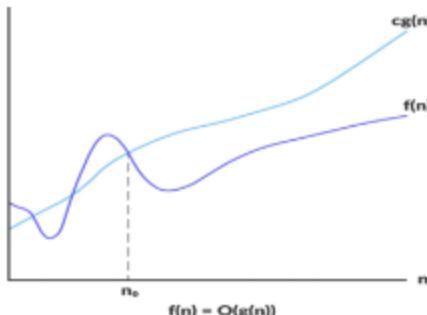
## O-notation : O - Big oh notation

- Suppose  $f(n)$  and  $g(n)$  can be any nonnegative functions defined on the set of natural numbers.
- $f(n)$  will be an **algorithm's running time** and  **$g(n)$**  will be some simple function to **compare** the count with.

37

## O-notation : O - Big oh notation

- **Formal definition:** A function  $f(n)$  is said to be in  $O(g(n))$ , denoted  $f(n) \in O(g(n))$ , if  $f(n)$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$ ,



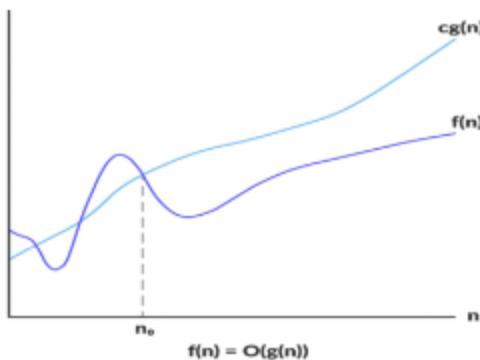
- i.e. if there exist some positive constant  $c$  and some nonnegative integer  $n_0$  such that:

$$f(n) \leq cg(n) \text{ for all } n \geq n_0$$

38

## O-notation : O - Big oh notation

The definition is illustrated in next figure.



Big-oh notation  $f(n) \in O(g(n))$ ,  
g(n) is an asymptotic upper bound for f (n).

This definition can be written as follow:

The function  $f(n)=O(g(n))$  iff  $\exists +ve$  constants  $c$  and  $n_0$   
Such that  $f(n) \leq c \times g(n) \forall n \geq n_0$

O = Asymptotic upper bound = Useful for worst case analysis

39

## Example (5)

Let  $f(n)=100n + 5$  prove that  $f(n) \in O(n)$

Proof:

$$\begin{aligned}100n + 5 &\leq 100n + 5n \quad (\text{for all } n \geq 1) \\&= 105n\end{aligned}$$

$$i.e. 100n + 5 \leq 105n$$

$$i.e. f(n) \leq cg(n)$$

$$\therefore 100n + 5 \in O(n) \text{ with } c=105 \text{ and } n_0=1$$

40

## Example (6)

Prove that  $2n + 3 \in O(n)$

Proof:

$$\text{Let } f(n) = 2n + 3$$

$$2n + 3 \leq \dots \quad (\text{single term only})$$

$$2n + 3 \leq 2n + 3n = 5n \quad n \geq 1$$

$$f(n) \leq c.g(n)$$

$$\therefore f(n) \in O(n)$$

$\therefore f(n) \in O(n)$  with  $c=5$  and  $n_0=1$

41

## Example (7)

Prove that the  $100n + 5 \in O(n^2)$

Proof:

$$\begin{aligned} 100n + 5 &\leq 100n + n \quad (\text{for all } n \geq 5) \\ &= 101n \\ &\leq 101n^2 \quad (\because n \leq n^2) \end{aligned}$$

Since, the definition gives us a lot of freedom in choosing specific values for constants  $c$  and  $n_0$ , we have  $c=101$  and  $n_0=5$

42

## No Uniqueness

- There is no unique set of values for  $n_0$  and  $c$  in proving the asymptotic bounds
- Prove that  $100n + 5 = O(n^2)$ 
  - (i)  $100n + 5 \leq 100n + n = 101n \leq 101n^2$  for all  $n \geq 5$   
You may pick  $n_0 = 5$  and  $c = 101$  to complete the proof.
  - (ii)  $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$  for all  $n \geq 1$   
You may pick  $n_0 = 1$  and  $c = 105$  to complete the proof.

43

## Example (8)

Prove that  $2n + 3 \in O(n^2)$

**Proof:**

$$2n + 3 \leq 2n^2 + 3n^2 = 5n^2 \quad n \geq 1$$

$$f(n) \leq c.g(n)$$

$\therefore f(n) \in O(n^2)$  with  $c = 5$  and  $n_0 = 1$

$\therefore f(n) \notin O(\log(n))$

**Note that:** Select the closest function is the best, the higher function is true but not useful.

$$1 < \log(n) < \sqrt{n} < n < n\log(n) < n^2 < n^3 < \dots < 2^n < 3^n < n^n$$

Lower bound      Average bound      Upper bound

44

## O-notation

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad \frac{1}{2}n(n-1) \in O(n^2).$$

$$n^3 \notin O(n^2), \quad 0.00001n^3 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2).$$

$$n^3 \in \Omega(n^2), \quad \frac{1}{2}n(n-1) \in \Omega(n^2), \quad \text{but } 100n + 5 \notin \Omega(n^2).$$

$$n^2 = O(n^2)$$

$$n^2 + n = O(n^2)$$

$$n^2 + 1000n = O(n^2)$$

$$5230n^2 + 1000n = O(n^2)$$

$$n = O(n^2)$$

$$\frac{n}{1200} = O(n^2)$$

$$n^{1.99999} = O(n^2)$$

$$\frac{n^2}{\log n} = O(n^2)$$

45

## Example (9)

- complexity of an algorithm

```
void f ( int a[], int n )
{
    int i;
    cout<< "N = "<< n;
    for ( i = 0; i < n; i++ )
        cout<<a[i];
    printf ( "n" );
}
```

2 \* O(1) + O(N)

O(N)

46

## Example (10)

- complexity of an algorithm

```
void f ( int a[], int n )
{
    int i;
    cout<< "N = "<< n;
    for ( i = 0; i < n; i++ )
        for (int j=0;j<n;j++)
            cout<<a[i]<<a[j];
    for ( i = 0; i < n; i++ )
        cout<<a[i];
    printf ( "n" );
}
```

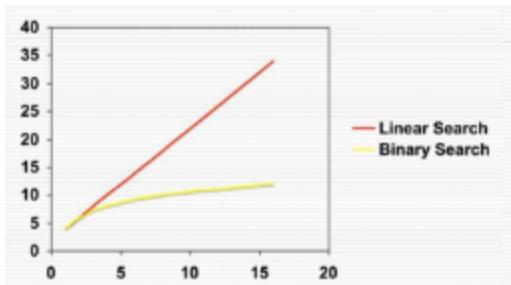
$2 * O(1) + O(N)+O(N^2)$

$O(N^2)$

47

## Example (11) : Linear search vs. binary search

- Linear Search
    - $- O(n).$
  - Binary Search**
    - $- O(\log_2 N)$
- | n       | $\log_2 n$ |
|---------|------------|
| 10      | 3          |
| 100     | 6          |
| 1,000   | 9          |
| 10,000  | 13         |
| 100,000 | 16         |



48

## Example (12) : Efficiency examples

Find the complexity of the following algorithm

```
statement1;  
statement2; } 3  
statement3;  
  
for (int i = 1; i <= N; i++) { } N  
    statement4;  
}  
  
for (int i = 1; i <= N; i++) { } 3N  
    statement5;  
    statement6;  
    statement7;  
}
```

49

## Example (13) : Efficiency examples

```
for (int i = 1; i <= N; i++) {  
    for (int j = 1; j <= N; j++) { } N2  
        statement1;  
    }  
}  
  
for (int i = 1; i <= N; i++) { } N2 + 4N  
    statement2;  
    statement3;  
    statement4;  
    statement5;
```

- How many statements will execute if  $N = 10$ ? If  $N = 1000$ ?

## Algorithm growth rates

- We measure runtime in proportion to the input data size, N.
  - growth rate: change in runtime as N changes.
- Say an algorithm runs  $0.4N^3 + 25N^2 + 8N + 17$  statements.
  - Consider the runtime when N is extremely large.
  - We ignore constants like 25 because they are tiny next to N.
  - The **highest-order term ( $N^3$ )** dominates the overall runtime.
- We say that this algorithm runs "**on the order of**"  $N^3$ .

51

## Complexity classes

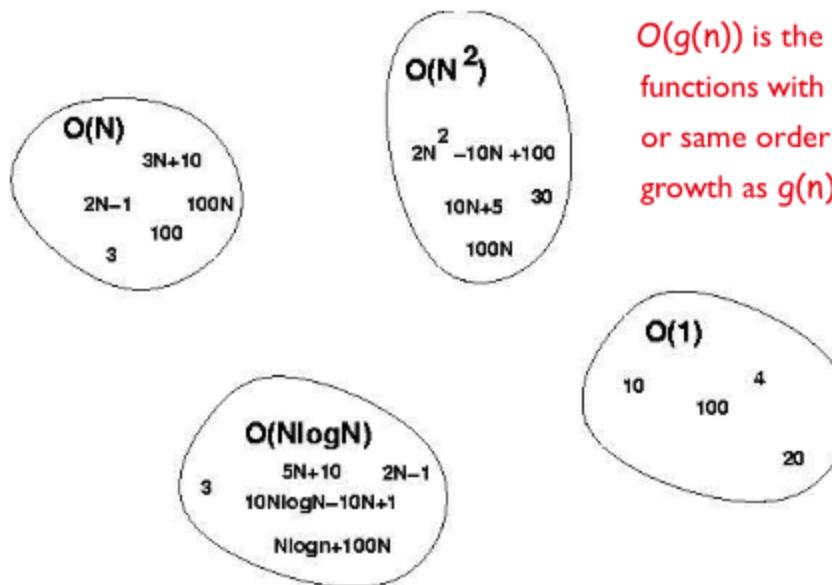
## Complexity classes

- **Complexity class:** A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	$O(1)$	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	$O(N)$	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

53

## Big-O Visualization



54

## Basic Efficiency classes

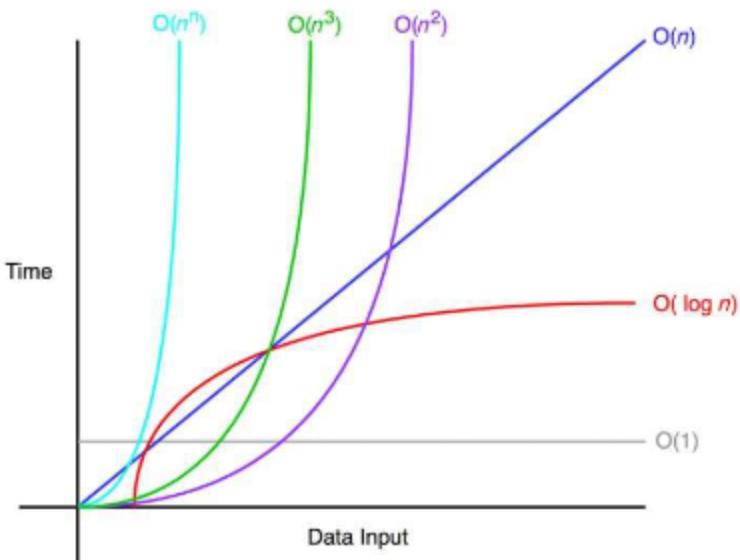
The time efficiencies of a large number of algorithms fall into only a few classes.

fast	1	constant	High time efficiency
	$\log n$	logarithmic	
	$n$	linear	
	$n \log n$	$n \log n$	
	$n^2$	quadratic	
	$n^3$	cubic	
	$2^n$	exponential	
slow	$n!$	factorial	

low time efficiency

55

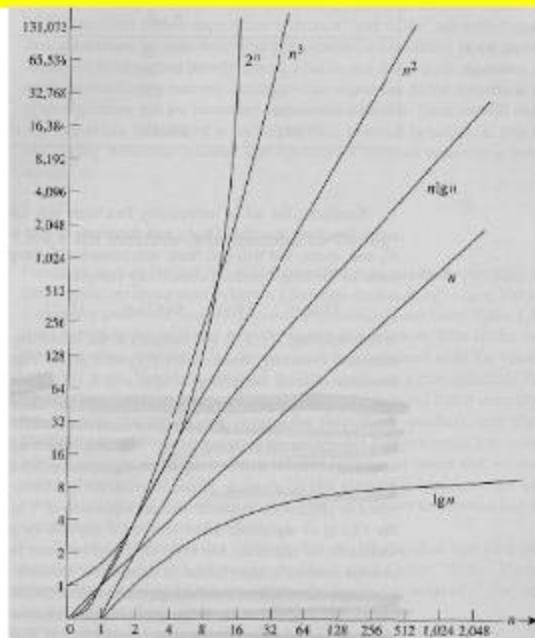
## Complexity classes



$$1 < \log(n) < \sqrt{n} < n < n \log(n) < n^2 < n^3 < \dots < 2^n < 3^n < n^n$$

56

## Complexity classes



$$1 < \log(n) < \sqrt{n} < n < n \log(n) < n^2 < n^3 < \dots < 2^n < 3^n < n^n$$

57

## Common orders of magnitude

**Table 1.4** Execution times for algorithms with the given time complexities

$n$	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 $\mu$ s*	0.01 $\mu$ s	0.033 $\mu$ s	0.1 $\mu$ s	1 $\mu$ s	$\mu$ s
20	0.004 $\mu$ s	0.02 $\mu$ s	0.086 $\mu$ s	0.4 $\mu$ s	8 $\mu$ s	ms†
30	0.005 $\mu$ s	0.03 $\mu$ s	0.147 $\mu$ s	0.9 $\mu$ s	27 $\mu$ s	s
40	0.005 $\mu$ s	0.04 $\mu$ s	0.213 $\mu$ s	1.6 $\mu$ s	64 $\mu$ s	18.3 mir
50	0.005 $\mu$ s	0.05 $\mu$ s	0.282 $\mu$ s	2.5 $\mu$ s	25 $\mu$ s	13 days
$10^2$	0.007 $\mu$ s	0.10 $\mu$ s	0.664 $\mu$ s	10 $\mu$ s	1 ms	$4 \times 10^{11}$ years
$10^3$	0.010 $\mu$ s	1.00 $\mu$ s	9.966 $\mu$ s	1 ms	1 s	
$10^4$	0.013 $\mu$ s	0 $\mu$ s	130 $\mu$ s	100 ms	16.7 min	
$10^5$	0.017 $\mu$ s	0.10 ms	1.67 ms	10 s	11.6 days	
$10^6$	0.020 $\mu$ s	1 ms	19.93 ms	16.7 min	31.7 years	
$10^7$	0.023 $\mu$ s	0.01 s	0.23 s	1.16 days	31,709 years	
$10^8$	0.027 $\mu$ s	0.10 s	2.66 s	115.7 days	$3.17 \times 10^7$ years	
$10^9$	0.030 $\mu$ s	1 s	29.90 s	31.7 years		

\*1  $\mu$ s =  $10^{-6}$  second.

†1 ms =  $10^{-3}$  second.

58

## Growth rate ranking of typical functions

$$f(n) = n^n$$

$$f(n) = 2^n$$

$$f(n) = n^3$$

$$f(n) = n^2$$

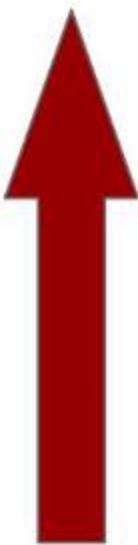
$$f(n) = n \log n$$

$$f(n) = n$$

$$f(n) = \sqrt{n}$$

$$f(n) = \log n$$

$$f(n) = 1$$



**grow fast**

**grow slowly**

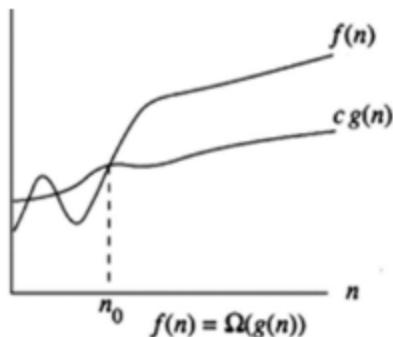
$$1 < \log(n) < \sqrt{n} < n < n \log(n) < n^2 < n^3 < \dots < 2^n < 3^n < n^n$$

**Big- $\Omega$  Notation**

## **Ω-notation**

### **Formal definition**

- A function  $f(n)$  is said to be in  $\Omega(g(n))$ , denoted  $f(n) \in \Omega(g(n))$ , if  $f(n)$  is **bounded below** by some **constant multiple of  $g(n)$**  for all **large  $n$** ,

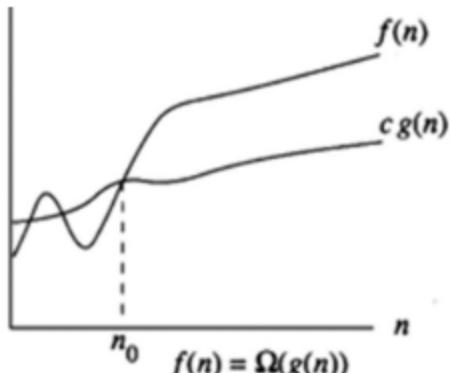


61

## **Ω-notation**

A function  $f(n)$  is said to be in  $\Omega(g(n))$ , if there exist some positive constant c and some nonnegative integer  $n_0$  such that

$$f(n) \geq cg(n) \text{ for all } n \geq n_0$$



$\Omega(g(n))$  is the set of functions with larger or same order of growth as  $g(n)$

62

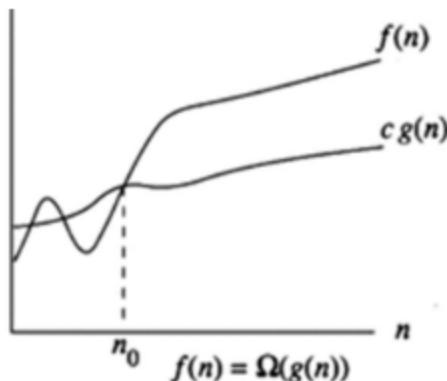
## $\Omega$ -notation

### Mathematical Definition:

The above definition can be written as follow:

The function  $f(n) = \Omega(g(n))$  iff  $\exists +ve$  constants  $c$  and  $n_0$

Such that  $f(n) \geq c \times g(n) \forall n \geq n_0$



$\Omega$  =Asymptotic lower bound = Useful for best case analysis

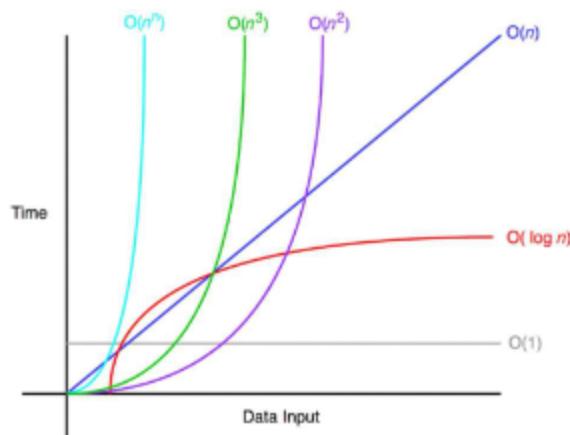
63

## Example

Here is an example for the formal proof that  $n^3 \in \Omega(n^2)$

$$n^3 \geq (n^2) \text{ for all } n \geq 0$$

i.e., we can select  $c=1$  and  $n_0=0$ .



64

## Example

Prove that  $n^3 + 10n^2 + 4n + 2 \in \Omega(n^2)$ .

Proof:

$$n^3 + 10n^2 + 4n + 2 \geq n^2 \text{ (for all } n \geq 0\text{)}$$

i.e., by definition  $f(n) \geq cg(n)$ , where  $c=1$  and  $n_0=0$

65

## Example

Let  $f(n) = 2n + 3$

Prove that

$$\begin{aligned}f(n) &= \Omega(n) \\f(n) &= \Omega(\log(n)) \\f(n) &\neq \Omega(n^2)\end{aligned}$$

**The answer**

$$2n + 3 \geq \dots \quad (\text{single term only})$$

$$2n + 3 \geq 1 * n \quad n \geq 1$$

$$f(n) \geq c.g(n)$$

$$\therefore f(n) = \Omega(n)$$

66

## Example

- $5n^2 = \Omega(n)$

$\exists c, n_0$  such that:

$$0 \leq cn \leq 5n^2$$

$$\Rightarrow cn \leq 5n^2$$

$$\Rightarrow c = 1 \text{ and } n > n_0 = 1$$

67

## Example

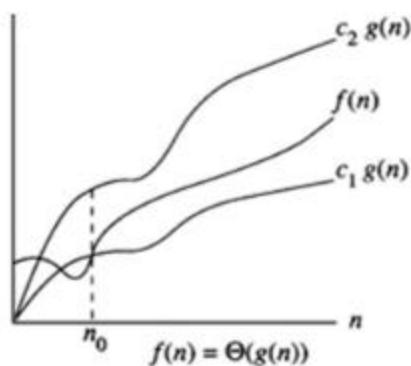
- $5n^2 = \Omega(n)$
- $n = \Omega(2n)$ ,
- $n^3 = \Omega(n^2)$ ,
- $n = \Omega(\log n)$
- $100n + 5 \neq \Omega(n^2)$

68

## Big- $\Theta$ Notation

### $\Theta$ -notation : Formal definition

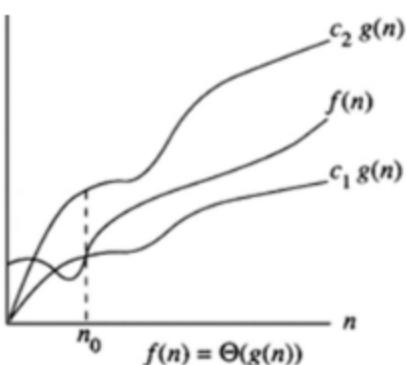
A function  $f(n)$  is said to be in  $\Theta(g(n))$ , denoted  $f(n) \in \Theta(g(n))$ , if  $f(n)$  is **bounded both above and below** by some positive constant multiples of  $g(n)$  for all large  $n$ .



## **Θ-notation : Formal definition**

A function  $f(n)$  is said to be in  $\Theta(g(n))$  if there exist some positive constant  $c_1$  and  $c_2$  and some nonnegative integer  $n_0$  such that

$$c_2 g(n) \leq f(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$



$\Theta(g(n))$  is the set of functions with the same order of growth as  $g(n)$

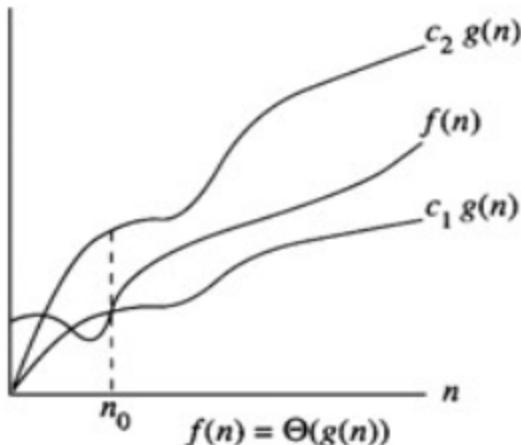
71

## **Θ-notation : Mathematical definition**

The above definition can be written as follow:

The function  $f(n) = \Theta(g(n))$  iff  $\exists +ve$  constants  $c_1$ ,  $c_2$  and  $n_0$

Such that  $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$  for all  $n \geq n_0$



72

## Example of $\Theta$ -notation

Prove that  $\frac{1}{2}n(n - 1) \in \Theta(n^2)$ .

Proof:

First prove the right inequality (the upper bound):

$$\frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \text{ for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \left[\frac{1}{2}n\right]\left[\frac{1}{2}n\right] \text{ for all } n \geq 2.$$

$$\therefore \frac{1}{2}n(n - 1) \geq \frac{1}{4}n^2$$

$$\text{i.e., } \frac{1}{4}n^2 \leq \frac{1}{2}n(n - 1) \leq \frac{1}{2}n^2$$

Hence,  $\frac{1}{2}n(n - 1) \in \Theta(n^2)$ , where  $c_2 = \frac{1}{4}$ ,  $c_1 = \frac{1}{2}$  and  $n_0 = 2$

73

## Exercises

prove the following using the above definition

- $10n^2 \in \Theta(n^2)$
- $10n^2 + 2n \in \Theta(n^2)$

74

## Example of $\Theta$ -notation

**Find a suitable  $g(n)$  such that  $f(n) = 2n + 3 \in \Theta(g(n))$**

**The answer:**

$$1 \times n \leq 2n + 3 \leq 5 \times n \text{ for all } n \geq 1$$

$$c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \text{ for all } n \geq n_0$$

$$\therefore f(n) = \Theta(n)$$

**Note that  $f(n) \neq \Theta(n^2)$**

**$f(n) = \Theta(n)$  only this is the correct**

75

## $\Theta$ -notation

**Note that:** asymptotic notation can be thought of as "relational operators" for functions similar to the corresponding relational operators for values.

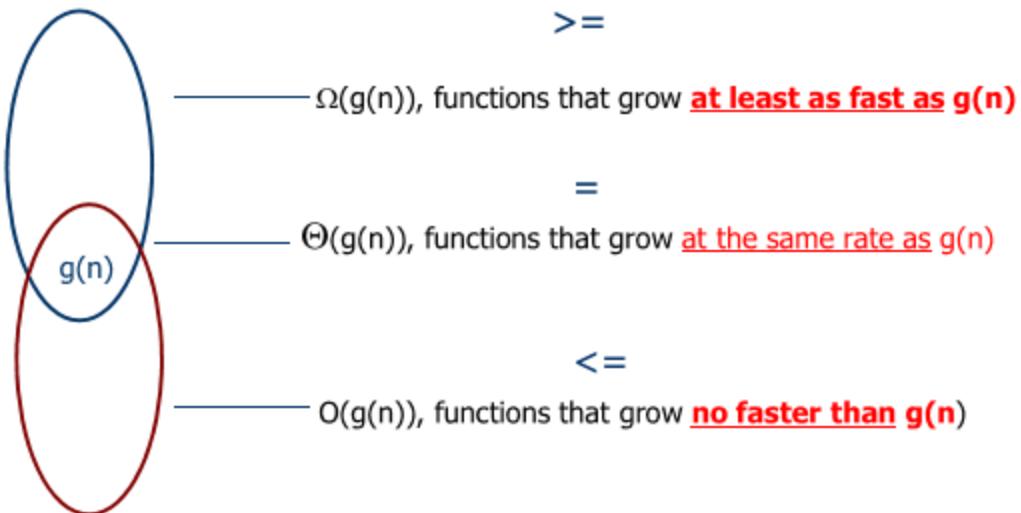
$$= \Rightarrow \Theta(\quad),$$

$$\leq \Rightarrow O(\quad),$$

$$\geq \Rightarrow \Omega(\quad)$$

76

## Θ-notation



77

## Examples

**Find a suitable  $g(n)$  such that  $f(n) = 2n^2 + 3n + 4 \in \Theta(g(n))$**

**Proof:**

$$\begin{aligned} 2n^2 + 3n + 4 &\leq 2n^2 + 3n^2 + 4n^2 \\ 2n^2 + 3n + 4 &\leq 9n^2 \quad \forall n \geq 1 \end{aligned}$$

i.e., by definition  $f(n) \leq cg(n)$ , where  $c=9$  and  $n_0 = 1$

$$\therefore f(n) \in O(n^2)$$

$$2n^2 + 3n + 4 \geq 1n^2$$

$$\therefore f(n) \in \Omega(n^2)$$

$\because f(n) \in O(n^2)$  and  $f(n) \in \Omega(n^2)$  then  $f(n) \in \Theta(n^2)$   
i.e.  $1n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$

78

## Examples

**Find a suitable g(n) such that  $f(n) = n^2 \log n + n \in \Theta(g(n))$**

$$1 \times n^2 \log n \leq n^2 \log n + n \leq 10 \times n^2 \log n$$

$$\therefore f(n) \in O(n^2 \log n) \text{ and } f(n) \in \Omega(n^2 \log n)$$

$$\text{then } f(n) \in \Theta(n^2 \log n)$$

79

## Examples

◦  $n^2/2 - n/2 = \Theta(n^2)$

$$\bullet \frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \quad \forall n \geq 0 \Rightarrow c_2 = \frac{1}{2}$$

$$\bullet \frac{1}{4} n^2 \leq \frac{1}{2} n^2 - \frac{1}{2} n \quad \forall n \geq 2 \Rightarrow c_1 = \frac{1}{4}$$

80

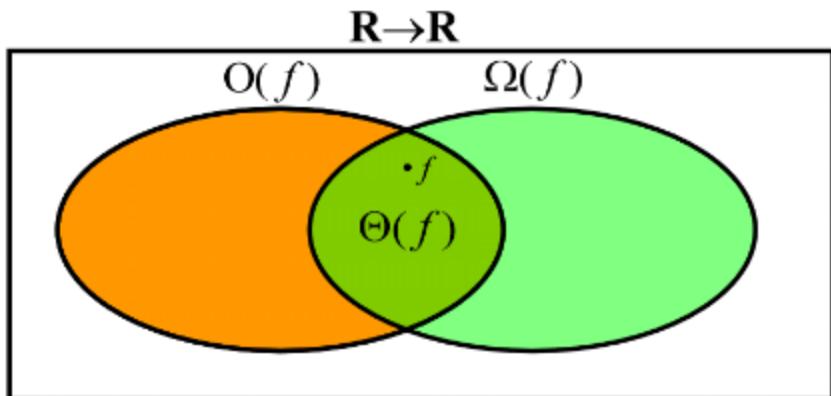
## Examples

- 1)  $n \neq \Theta(n^2)$ :  $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$  only holds for:  $n \leq 1/c_1$
- 2)  $6n^3 \neq \Theta(n^2)$ :  $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$  only holds for:  $n \leq c_2 / 6$
- 3)  $n \neq \Theta(\log n)$

81

## Relations Between Different Sets

- Subset relations between order-of-growth sets.



82

# Logarithms and properties

- In algorithm analysis we often use the notation “**log n**” without specifying the base

<b>Binary logarithm</b>	$\lg n = \log_2 n$	$\log x^y = y \log x$
<b>Natural logarithm</b>	$\ln n = \log_e n$	$\log xy = \log x + \log y$
		$\log \frac{x}{y} = \log x - \log y$
	$\lg^k n = (\lg n)^k$	$a^{\log_b x} = x^{\log_b a}$
	$\lg \lg n = \lg(\lg n)$	$\log_b x = \frac{\log_a x}{\log_a b}$

83

## More Examples

- For each of the following pairs of functions, either  $f(n)$  is  $O(g(n))$ ,  $f(n)$  is  $\Omega(g(n))$ , or  $f(n) = \Theta(g(n))$ . Determine which relationship is correct.

◦ $f(n) = \log n^2$ ; $g(n) = \log n + 5$	$f(n) = \Theta(g(n))$
◦ $f(n) = n$ ; $g(n) = \log n^2$	$f(n) = \Omega(g(n))$
◦ $f(n) = \log \log n$ ; $g(n) = \log n$	$f(n) = O(g(n))$
◦ $f(n) = n$ ; $g(n) = \log^2 n$	$f(n) = \Omega(g(n))$
◦ $f(n) = n \log n + n$ ; $g(n) = \log n$	$f(n) = \Omega(g(n))$
◦ $f(n) = 10$ ; $g(n) = \log 10$	$f(n) = \Theta(g(n))$
◦ $f(n) = 2^n$ ; $g(n) = 10n^2$	$f(n) = \Omega(g(n))$
◦ $f(n) = 2^n$ ; $g(n) = 3^n$	$f(n) = O(g(n))$

84

## Summary : Asymptotic Notations

Assuming  $f(n)$ ,  $g(n)$  and  $h(n)$  be asymptotic functions the mathematical definitions are:

1. If  $f(n) = \Theta(g(n))$ , then there exists positive constants  $c_1, c_2, n_0$  such that  $0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$ , for all  $n \geq n_0$
2. If  $f(n) = O(g(n))$ , then there exists positive constants  $c, n_0$  such that  $0 \leq f(n) \leq c.g(n)$ , for all  $n \geq n_0$
3. If  $f(n) = \Omega(g(n))$ , then there exists positive constants  $c, n_0$  such that  $0 \leq c.g(n) \leq f(n)$ , for all  $n \geq n_0$

85

## Properties of Asymptotic Notations

$O$ ,  $\Omega$  and  $\Theta$

## Properties of Asymptotic Notations

As we have gone through the definition of these three notations (Big-O, Omega- $\Omega$ , Theta- $\Theta$ ) in our previous section. Now let's discuss some important properties of those notations.

### Properties of Asymptotic Notations

1. General Properties
2. Transitive Properties
3. Symmetric Properties

87

### 1) General Properties

If  $f(n)$  is  $O(g(n))$  **then**  $a*f(n)$  is also  $O(g(n))$ ; where  $a$  is a constant.

#### Example:

$f(n) = 2n^2+5$  is  $O(n^2)$

then  $7*f(n) = 7(2n^2+5) = 14n^2+35$  is also  $O(n^2)$

88

## 1) General Properties

Similarly this property satisfies for both  $\Theta$  and  $\Omega$  notation.

We can say:

- If  $f(n)$  is  $\Theta(g(n))$  then  $a*f(n)$  is also  $\Theta(g(n))$ ; where  $a$  is a constant.
- If  $f(n)$  is  $\Omega(g(n))$  then  $a*f(n)$  is also  $\Omega(g(n))$ ; where  $a$  is a constant.

Similarly  $\Omega$  And  $\Theta$  also have the same properties

89

## 2) Transitivity

If  $f(n) \in O(g(n))$  and  $g(n) \in O(h(n))$  then  $f(n) \in O(h(n))$ .

Note similarity with  $a \leq b$

**Example:**

If  $f(n) = n$ ,  $g(n) = n^2$  and  $h(n) = n^3$

$\Rightarrow n$  is  $O(n^2)$  and  $n^2$  is  $O(n^3)$  then  $n$  is  $O(n^3)$

90

## 2) Transitivity

Similarly this property satisfies for both  $\Theta$  and  $\Omega$  notation.

**We can say:**

1. If  $f(n)$  is  $\Theta(g(n))$  and  $g(n)$  is  $\Theta(h(n))$  then  $f(n) = \Theta(h(n))$ .
2. If  $f(n)$  is  $\Omega(g(n))$  and  $g(n)$  is  $\Omega(h(n))$  then  $f(n) = \Omega(h(n))$

91

## 3) Symmetric properties

If  $f(n) \in \Theta(g(n))$  then  $g(n) \in \Theta(f(n))$ .

**Example:**

$f(n) = n^2$  and  $g(n) = n^2$  then  $f(n) = \Theta(n^2)$  and  $g(n) = \Theta(n^2)$

This property only satisfies for  $\Theta$  notation **only**.

92

## Some More Properties

1. If  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$  then  $f(n) = \Theta(g(n))$

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

$$\therefore f(n) = \theta(g(n))$$

93

## Some More Properties

Useful Property Involving the Asymptotic Notation.

The following property, in particular, is useful in analyzing algorithms that **comprise two consecutively** executed parts.

### THEOREM

2. If  $f_1(n) \in O(g_1(n))$  and  $f_2(n) \in O(g_2(n))$ , then

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

The analogous assertions are true for the  $\Omega$ -notation and  $\Theta$ -notation.

94

## Some More Properties

### Example:

$$f_1(n) = n \quad \text{i.e. } O(n)$$

$$f_2(n) = n^2 \quad \text{i.e. } O(n^2)$$

$$\text{then } f_1(n) + f_2(n) = n + n^2 \text{ i.e. } f_1(n) + f_2(n) \in O(\max\{n, n^2\}) = O(n^2)$$

**Implication:** The algorithm's overall efficiency will be determined by the part with a **larger order** of growth.

- For example,

$$- 5n^2 + 3n\log n \in O(n^2)$$

95

## Some More Properties

If  $f_1(n) \in O(g_1(n))$  and  $f_2(n) \in O(g_2(n))$ , then

$$f_1(n) * f_2(n) \in O(g_1(n)*g_2(n))$$

### Example:

$$f_1(n) = n \quad \text{i.e. } O(n)$$

$$f_2(n) = n^2 \quad \text{i.e. } O(n^2)$$

$$\text{then } f_1(n) * f_2(n) = n * n^2 = n^3 \quad \text{i.e. } O(n^3)$$

96

## Using Limits for Comparing Orders of Growth

$$\lim_{n \rightarrow \infty} f(n)/g(n) = \begin{cases} 0 & \text{order of growth of } f(n) < \text{order of growth of } g(n) \\ c & \text{order of growth of } f(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } f(n) > \text{order of growth of } g(n) \end{cases}$$

97

## L'Hôpital's rule

If  $\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$  and the derivatives  $f'$ ,  $g'$  exist,  
Then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

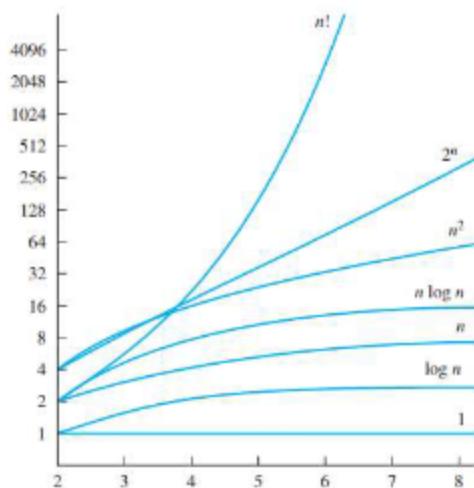
98

## THEOREM 1

Let  $f(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$ , where  $a_0, a_1, \dots, a_{n-1}, a_n$  are real numbers. Then  $f(x)$  is  $O(x^n)$ .

99

## The Growth of Functions Commonly Used in Big-O Estimates



## The Growth of Functions Commonly Used in Big-O Estimates.

100

## Assignments

### Exercise (1) : (True or False)

If  $f(n) = \Theta(g(n))$  and  $g(n) = \Theta(h(n))$ , then  $h(n) = \Theta(f(n))$

**Solution:** True.  $\Theta$  is transitive.

## Exercise (2) : (True or False)

If  $f(n) = O(g(n))$  and  $g(n) = O(h(n))$ , then  $h(n) = \Omega(f(n))$

**Solution:** True.  $O$  is transitive, and  $h(n) = \Omega(f(n))$  is the same as  $f(n) = O(h(n))$

103

## Exercise (3) : (True or False)

$$\frac{n}{100} = \Omega(n)$$

**Solution:** True.  $\frac{n}{100} < c * n$  for  $c = \frac{1}{200}$ .

104

## Exercise (4)

Which kind of growth best characterizes each of these functions?

	Constant	Linear	Polynomial	Exponential
$(3/2)^n$	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
1	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
$(3/2)n$	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
$2n^3$	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
$2^n$	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
$3n^2$	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
1000	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
$3n$	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>

105

## Exercise (5)

Rank these functions according to their growth, from slowest growing (at the top) to fastest growing (at the bottom).

Answer

$n^2$	1
$n$	$n$
$2^n$	$n^2$
$(3/2)^n$	$n^3$
$n^3$	$(3/2)^n$
1	$2^n$

106

## Exercise (6)

Rank these functions according to their growth, from slowest growing to fastest growing.

Answer

$n \log_6 n$

64

$\log_2 n$

$\log_8 n$

$\log_5 n$

$\log_2 n$

$8n^2$

$4n$

$n \log_2 n$

$n \log_6 n$

$64$

$n \log_2 n$

$6n^3$

$8n^2$

$8^{2n}$

$6n^3$

107

## Exercise (7)

Match each function with an equivalent function, in terms of their  $\Theta$ .

Only match a function if  $f(n)=\Theta(g(n))$ .

Answer

$f(n)$	$g(n)$
$n + 30$	$n^4$
$n^2 + 2n - 10$	$3n - 1$
$n^3 * 3n$	$\log_2 2x$
$\log_2 x$	$n^2 + 3n$

$f(n)$	$g(n)$
$n + 30$	$3n - 1$
$n^2 + 2n - 10$	$n^2 + 3n$
$n^3 * 3n$	$n^4$
$\log_2 x$	$\log_2 2x$

108

## Exercise (8)

For the functions,  $\log_2 n$  and  $\log_8 n$ , what is the asymptotic relationship between these functions?

Choose all answers that apply:

(A)  $\log_2 n$  is  $O(\log_8 n)$

(B)  $\log_2 n$  is  $\Omega(\log_8 n)$

(C)  $\log_2 n$  is  $\Theta(\log_8 n)$

Answer

✓ CORRECT (SELECTED)

$\log_2 n$  is  $O(\log_8 n)$

✓ CORRECT (SELECTED)

$\log_2 n$  is  $\Omega(\log_8 n)$

✓ CORRECT (SELECTED)

$\log_2 n$  is  $\Theta(\log_8 n)$

109

## **Section No. 06**

### **Answer the following Questions.**

#### **1. Show that**

$f(x) = x^2 + 2x + 1$  is  $O(x^2)$ .

$7x^2$  is  $O(x^3)$ .

$n^2$  is not  $O(n)$ .

$3x^2 + 8x \log x$  is  $\Theta(x^2)$ .

#### **2. Show that**

$7x^2$  is  $O(x^3)$ . Is it also true that  $x^3$  is  $O(7x^2)$ ?

3. How can Big-O notation be used to estimate the sum of the first n positive integers?
4. Give a Big-O estimate for

$$f(x) = (x + 1) \log(x^2 + 1) + 3x^2.$$

5. Determine whether each of these functions is  $O(x)$ .

- a)  $f(x) = 10$       b)  $f(x) = 3x + 7$   
c)  $f(x) = x^2 + x + 1$       d)  $f(x) = 5 \log x$

6. Determine whether each of these functions is  $O(x^2)$

- a)  $f(x) = 17x + 11$       b)  $f(x) = x^2 + 1000$   
c)  $f(x) = x \log x$       d)  $f(x) = x^4/2$

7.

Use the definition of “ $f(x)$  is  $O(g(x))$ ” to show that  $x^4 + 9x^3 + 4x + 7$  is  $O(x^4)$ .

8.

. Find the least integer  $n$  such that  $f(x)$  is  $O(x^n)$  for each of these functions.

- a)  $f(x) = 2x^3 + x^2 \log x$
- b)  $f(x) = 3x^3 + (\log x)^4$
- c)  $f(x) = (x^4 + x^2 + 1)/(x^3 + 1)$
- d)  $f(x) = (x^4 + 5 \log x)/(x^4 + 1)$

9.

Show that  $x^3$  is  $O(x^4)$  but that  $x^4$  is not  $O(x^3)$ .

10.

. Give as good a big- $O$  estimate as possible for each of these functions.

- a)  $(n^2 + 8)(n + 1)$
- b)  $(n \log n + n^2)(n^3 + 2)$
- c)  $(n! + 2^n)(n^3 + \log(n^2 + 1))$

# **Chapter (7)**

## **Recursive and Generating Functions**

## Outline

---

1. Sequences
  2. Closed formula
  3. Recursive Definitions.
  4. Geometric Sequences
  5. Generating Functions
- 

5

## Sequences

## Describing Sequences

---

- A **sequence** is simply **an ordered list of numbers**.
- For example, here is a **sequence**: 0, 1, 2, 3, 4, 5, ...
- we use variables to represent **terms** in a sequence  
they will look like this:

$$a_0, a_1, a_2, a_3, \dots$$

- To refer to the entire sequence at once, we will write

$$(a_n)_{n \in N} \text{ or } (a_n)_{n \geq 0}$$

---

7

## Describing Sequences

---

- Another way to specify a sequence.
- We consider two ways to do this:
  1. **Closed formula.**
  2. **Recursive definition.**

---

8

## Closed formula

### Closed formula.

---

- A **closed formula** for a sequence  $(a_n)_{n \in N}$  is a **formula** for  $a_n$  using a **fixed finite number** of **operations** on  $n$ .
- This is what you normally think of as a **formula** in  $n$ , just as if you were **defining a function** in **terms** of  $n$ .

## Example (1)

---

Here are a few **closed formulas** for sequences:

$$a_n = n^2.$$

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

$$a_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^{-n}}{\sqrt{5}}.$$

---

11

## Example (1)

---

- Note in each formula, if you are given  $n$ , you can calculate  $a_n$  **directly**: just **plug** in  $n$ .
- For example, to find  $a_3$  in the second sequence, just compute

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

$$a_3 = \frac{3(3+1)}{2} = 6.$$

---

12

## Common closed formula

1, 4, 9, 16, 25, ...

The **square numbers**. The sequence  $(s_n)_{n \geq 1}$  has closed formula  $s_n = n^2$

1, 3, 6, 10, 15, 21, ....

The **triangular numbers**. The sequence  $(T_n)_{n \geq 1}$  has closed formula  $T_n = \frac{n(n+1)}{2}$

1, 2, 4, 8, 16, 32, ...

The powers of 2. The sequence  $(a_n)_{n \geq 0}$  with closed formula  $a_n = 2^n$ .

---

13

## Recursive Definitions

## Recursive Definitions

- Sometimes it is difficult to define an object **explicitly using closed formula.**
- However, it may be **easy** to define this object in terms of **itself**.
- This process is called **recursion**.

**Recursion:** The process of defining an object in terms of itself.

---

15

## Recursive definition.

- A **recursive definition** for a sequence  $(a_n)_{n \in N}$  consists of
  1. **A recurrence relation :** an equation relating a term of the sequence to **previous terms** (terms with smaller index) and
  2. **An initial condition:** a list of a **few terms** of the sequence (one less than the number of terms in the recurrence relation).

---

16

## Example (2)

---

- Here are a few **recursive definitions** for sequences:

$a_n = 2a_{n-1}$  with  $a_0 = 1$ .

$a_n = 2a_{n-1}$  with  $a_0 = 27$ .

$a_n = a_{n-1} + a_{n-2}$  with  $a_0 = 0$  and  $a_1 = 1$ .

- In these formulas, if you are given  $n$ , you **cannot calculate an directly**, you first need to find  $a_{n-1}$  (or  $a_{n-1}$  and  $a_{n-2}$ ).
- 

17

## Example (2)

---

$a_n = 2a_{n-1}$  with  $a_0 = 27$ .

- In the second sequence, to find  $a_3$
- you would take  $2a_2$ , but to find  $a_2 = 2a_1$  we would need to know  $a_1 = 2a_0$ .
- We do know this, so we could trace back through these equations to find  $a_1 = 54$ ,  $a_2 = 108$  and finally  $a_3 = 216$
- 

18

## Closed formula vs recursive definition

### Closed formula vs recursive definition

- You might wonder why we would **bother** with recursive definitions for sequences.
- After all, it is **harder** to find  $a_n$  with a **recursive definition** than with a **closed formula**.
- This is true, but it is **also harder** to find a **closed formula** for a sequence than it is to **find a recursive definition**.

## Recursive Definitions

---

- So to find a **useful closed formula**, we might **first** find the **recursive definition**, **then** use that to find the **closed formula**.

---

21

## Recursive Definitions

---

- This is not to say that **recursive definitions aren't** useful in finding  $a_n$ .
- You can always calculate  $a_n$  given a recursive definition, it might just take a while.

---

22

## Example (3)

Find  $a_6$  in the sequence defined by  $a_n = 2a_{n-1} - a_{n-2}$  with  $a_0 = 3$  and  $a_1 = 4$ .

Solution:

- We know that  $a_6 = 2a_5 - a_4$ . So to find  $a_6$  we need to find  $a_5$  and  $a_4$ . Well

$$a_5 = 2a_4 - a_3 \quad \text{and} \quad a_4 = 2a_3 - a_2,$$

- so if we can only find  $a_3$  and  $a_2$  we would be set. Of course

$$a_3 = 2a_2 - a_1 \quad \text{and} \quad a_2 = 2a_1 - a_0,$$

23

## Example (3)

- so we only need to find  $a_1$  and  $a_0$ . But we are given these. Thus

$$a_0 = 3$$

$$a_1 = 4$$

$$a_2 = 2 \cdot 4 - 3 = 5$$

$$a_3 = 2 \cdot 5 - 4 = 6$$

$$a_4 = 2 \cdot 6 - 5 = 7$$

$$a_5 = 2 \cdot 7 - 6 = 8$$

$$a_6 = 2 \cdot 8 - 7 = 9.$$

24

## Example (3)

---

- Note that now we can **guess a closed formula** for the nth term of the sequence:  $a_n = n + 3$ .
- To be sure this will always work, we could plug in this formula into the recurrence relation:

$$\begin{aligned}2a_{n-1} - a_{n-2} &= 2((n-1) + 3) - ((n-2) + 3) \\&= 2n + 4 - n - 1 \\&= n + 3 = a_n.\end{aligned}$$

---

25

## Example (3)

---

- That is not quite enough though, since there can be **multiple closed formulas** that satisfy the same recurrence relation; we must also **check** that our closed formula agrees on the **initial terms** of the sequence.
- Since  $a_0 = 0 + 3 = 3$  and  $a_1 = 1 + 3 = 4$  are the correct initial conditions, we can now conclude we have the **correct closed formula**.

---

26

## Example (4)

---

1, 1, 2, 3, 5, 8, 13, ...

The **Fibonacci numbers** (or Fibonacci sequence), defined recursively by  $F_n = F_{n-1} + F_{n-2}$  with  $F_1 = F_2 = 1$ .

---

27

## Recursively Defined Functions

---

- We use two steps to define a function with the set of nonnegative integers as its domain:
- **BASIS STEP:** Specify the value of the function at the first point.
- **RECURSIVE STEP:** Specifying how terms in the function are found from previous terms.

---

28

## Example (5)

The sequence of powers of 2 is given by  $a_n = 2^n$  for  $n = 0, 1, 2, \dots$

Solution:

Explicit Formula

Basis Step:

Specify the value of the sequence at zero.

$$a_0 = 2^0 = 1$$

Recursive Step:

Give a rule for finding a term of the sequence from the previous one.

$$a_{n+1} = 2a_n, \quad \text{for } n = 0, 1, 2, \dots$$

Recursive Formula

---

29

## Example (6)

Suppose that  $f$  is defined recursively by

$$f(0) = 3,$$

$$f(n + 1) = 2f(n) + 3.$$

Find  $f(1)$ ,  $f(2)$ ,  $f(3)$ , and  $f(4)$ .

Solution:

From the recursive definition it follows that

$$f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9,$$

$$f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21,$$

$$f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45,$$

$$f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93.$$

---

30

## Example (7)

Give a recursive definition of the factorial function  $n!$

Solution:

**Basis Step:**

Specify the value of the function at zero.

$$f(0) = 1$$

**•Recursive Step:**

Give a rule for finding its value at an integer from its values at smaller integers.

$$f(n + 1) = (n + 1) \cdot f(n) , \quad \text{for } n = 0, 1, 2, \dots$$

---

31

## Example (8)

The Fibonacci numbers,  $f_0, f_1, f_2, \dots$ , are defined by the equations  $f_0 = 0$ ,  $f_1 = 1$ , and  $f_n = f_{n-1} + f_{n-2}$

Find  $f_2, f_3, f_4, f_5$

Solution:

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$

$$f_3 = f_2 + f_1 = 1 + 1 = 2$$

$$f_4 = f_3 + f_2 = 2 + 1 = 3$$

$$f_5 = f_4 + f_3 = 3 + 2 = 5$$

---

32

## Geometric sequences

### Geometric Sequences

- A sequence is called **geometric** if the ratio between successive terms is constant. Suppose the initial term  $a_0$  is a and the **common ratio** is r.
- Then we have,
  1. **Closed formula:**  $a_n = a \cdot r^n$ .  $n = 0, 1, 2, 3, \dots$
  2. **Recursive definition:**  $a_n = r a_{n-1}$  with  $a_0 = a$ .  $n = 1, 2, 3, \dots$

## Geometric sequences

A *geometric progression* is a sequence of the form

$$a, ar, ar^2, \dots, ar^n, \dots$$

where the *initial term a* and  
the *common ratio r* are real numbers.

$$2, 10, 50, 250, \dots$$

---

35

### Example (8)

$$1, -1, 1, -1, 1, \dots;$$

$$\{ar^n\}, \quad n = 0, 1, 2, \dots$$

$$a = 1$$

$$r = -1$$

---

36

### Example (9)

---

$2, 10, 50, 250, 1250, \dots;$

$$\{ar^n\}, \quad n = 0, 1, 2, \dots$$

$$a = 2$$

$$r = 5$$

---

37

### Example (10)

---

$6, 2, \frac{2}{3}, \frac{2}{9}, \frac{2}{27}, \dots$

$$\{ar^n\}, \quad n = 0, 1, 2, \dots$$

$$a = 6$$

$$r = 1/3$$

---

38

## Example (11)

---

Find  $a, r$ ?  $\{3 * 4^n\}, \ n = 0, 1, 2, \dots$

$$\{ar^n\}, \quad n = 0, 1, 2, \dots$$

$$a = 3$$

$$r = 4$$

---

39

## Example (12)

---

Find  $a, r$ ?  $\{3 * 4^n\}, \ n = 1, 2, 3, \dots$

$$a = 12$$

$$r = 4$$

---

40

## Example (13)

---

- Find the **recursive and closed formula** for the geometric sequences below. Again, the first term listed is  $a_0$ .
- 3, 6, 12, 24, 48, ....
  - 27, 9, 3, 1,  $\frac{1}{3}$ , ....

Solution:

---

41

## Example (13)

---

Solution:

- Start by checking that these sequences really are geometric by dividing each term by its previous term. If this ratio really is constant, we will have found  $r$ .

---

42

## Example (13)

Solution:

- 3, 6, 12, 24, 48, ....
- $6/3 = 2$ ,  $12/6 = 2$ ,  $24/12 = 2$ , etc.
- Yes, to get from any term to the next, we multiply by  $r=2$ .
- So the
  1. **Recursive definition** is  $a_n = 2a_{n-1}$  with  $a_0=3$ .
  2. **The closed formula** is  $a_n = 3 \cdot 2^n$ .

---

43

## Example (13)

- 27, 9, 3, 1,  $1/3$ , ....
- The common ratio is  $r = 1/3$ .
- So the sequence has
  1. **recursive definition**  $a_n = \frac{1}{3}a_{n-1}$  with  $a_0=27$  and
  2. **closed formula**  $a_n = 27 \cdot (\frac{1}{3})^n$

---

44

## Sum of infinite geometric series

### Sum of infinite geometric series

$$S_{\infty} = \left\{ \frac{a}{1-r}, -1 < r < 1 \right.$$

where  $a$  = common ratio,  $a$  = first term of the series

$\sum_{n=1}^{\infty} a_1(r)^{n-1}$  converges if and only if  $-1 < r < 1$

$$\sum_{n=1}^{\infty} a_1(r)^{n-1} = \frac{a_1}{1-r}$$

---

45

Generating Functions

## Generating Functions

- Let that **an infinite sequence** 2, 3, 5, 8, 12, .....
- Instead of **the infinite** we look at **a single function** which **encodes** the sequence.
- A function whose **power series** displays the terms of the sequence.
- So for example, we would look at the **power series**  
 **$G(x)=2+3x+5x^2+8x^3+12x^4+\dots$**  which displays the sequence **2, 3, 5, 8, 12, .....** as **coefficients**.

47

## generating function

- Let that **an infinite sequence**

$$a_0, a_1, a_2, \dots$$

- Then the **generating function is**

$$G(x) = a_0 + a_1 x + \dots + a_n x^n + \dots$$

$$G(x) = \sum_{k=0}^{\infty} a_k x^k$$

48

## Generating Functions

**Def 1.** The **generating function** for the sequence

$a_0, a_1, a_2, \dots$  of real numbers is the infinite series

$$G(x) = a_0 + a_1 x + \dots + a_n x^n + \dots$$

$$= \sum_{k=0}^{\infty} a_k x^k$$

---

49

### Example (14)

What sequence is represented by the generating function

$$3 + 8x^2 + x^2 + \frac{x^5}{7} + 100x^6 + \dots ?$$

**Solution:**

We just read off the coefficients of each  $x^n$  term.

$$a_0 = 3 \quad a_1 = 0 \quad a_2 = 8,$$

$$a_3 = 1, a_4 = 0, \text{ and } a_5 = \frac{1}{7}.$$

So we have the sequence

$$3, 0, 8, 1, 0, \frac{1}{7}, 100, \dots$$

---

50

## Building Generating Functions

### Building Generating Functions

- Our goal now is to **gather** some tools to build the **generating function** of a particular given sequence.
- Let's see what the generating functions are for some very **simple** sequences.

## Example (15)

- The simplest of all: 1, 1, 1, 1, 1, . . .
- What does the generating series look like?
- It is simply  $G(x) = 1 + x + x^2 + x^3 + x^4 + \dots$
- Now, **can we find a closed formula for this power series?**
- Yes!

53

## Example (15)

- This particular series is really just a **geometric series** with common **ratio  $x$** . So if we use our “**multiply, shift and subtract**” technique, we have

$$\begin{array}{rcl} S &= & 1 + x + x^2 + x^3 + \dots \\ - & xS &= & x + x^2 + x^3 + x^4 + \dots \\ \hline (1-x)S &= & 1 \end{array}$$

- Therefore we see that

$$G(x) = 1 + x + x^2 + x^3 + x^4 + \dots = \frac{1}{1-x}$$

- This is only true on the **interval of convergence** for the power series, in this case when  $|x| < 1$ .

54

## Note that

The generating function for 1, 1, 1, 1, 1, ... is  $G(x) = \frac{1}{1-x}$

$$G(x) = \frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

How to find generating functions

Using  $\frac{1}{1-x}$

## Finding generating functions

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

How to find generating functions from  $\frac{1}{1-x}$  using :

1. substitution,
2. multiplication (by a constant or by  $x$ ),
3. addition, and
4. differentiation.

To use each of these, you must **notice** a way to **transform** the sequence  $1, 1, 1, 1, 1, \dots$  into your **desired sequence**.

57

## Finding generating functions

- Let's use this **basic generating function** to find generating functions for **more sequences**.

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

58

## Example (16) : substitution

---

- What if we **replace x by -x**.

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

- We get

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 + \dots \text{ which generates } 1, -1, 1, -1, \dots$$

---

59

## Example (17) : substitution

---

- If we **replace x by 3x**

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

- We get

$$\frac{1}{1-3x} = 1 + 3x + 9x^2 + 27x^3 + \dots \text{ which generates } 1, 3, 9, 27, \dots$$

---

60

## Note that (\*)

- By **replacing** the  $x$  in  $\frac{1}{1-x}$  we can get generating functions for a variety of sequences, **but not all**.
- For example, **you cannot plug** in anything for  $x$  to get the generating function for **2, 2, 2, 2, ....**
- However, we are **not lost yet**.
- Notice that each term of 2, 2, 2, 2, .... is the result of **multiplying** the terms of 1, 1, 1, 1, .... by the **constant 2**.

61

## Example (18) : multiplication

So **multiply** the generating function by **2** as well.

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

- We get

$$\frac{2}{1-x} = 2 + 2x + 2x^2 + 2x^3 + 2x^4 + \dots \text{ which generates } 2, 2, 2, 2, \dots$$

62

## Example (19) : multiplication

Find the generating function for the sequence

$$3, 9, 27, 81, \dots,$$

Solution:

- we note that this sequence is the result of multiplying each term of 1, 3, 9, 27, ..... by 3.
- Since we have the generating function for 1, 3, 9, 27, ..

$$\frac{1}{1-3x} = 1 + 3x + 9x^2 + 27x^3 + \dots \text{ which generates } 1, 3, 9, 27, \dots$$

63

## Example (20) : multiplication

So **multiply** the generating function by **3**.

$$\frac{1}{1-3x} = 1 + 3x + 9x^2 + 27x^3 + \dots \text{ which generates } 1, 3, 9, 27, \dots$$

- we can say

$$\frac{3}{1-3x} = 3 \cdot 1 + 3 \cdot 3x + 3 \cdot 9x^2 + 3 \cdot 27x^3 + \dots$$

which generates **3, 9, 27, 81, ...**

64

## Example (21) : addition

Find the generating function for the sequence

$$2, 4, 10, 28, 82, \dots$$

Solution:

- Here the terms are always 1 more than powers of 3.
- That is, we have added the sequences  $1, 1, 1, 1, \dots$  and  $1, 3, 9, 27, 81, \dots$  term by term.

---

65

## Example (22) : addition

- Therefore we can get a generating function by **adding** the respective generating functions:

$$\frac{1}{1-3x} = 1 + 3x + 9x^2 + 27x^3 + \dots \text{ which generates } 1, 3, 9, 27, \dots$$

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

$$\begin{aligned}2 + 4x + 10x^2 + 28x^3 + \dots &= (1+1) + (1+3)x + (1+9)x^2 + (1+27)x^3 + \dots \\&= 1 + x + x^2 + x^3 + \dots + 1 + 3x + 9x^2 + 27x^3 + \dots \\&= \frac{1}{1-x} + \frac{1}{1-3x}\end{aligned}$$

---

## Example (23) : Multiplying by x (*shift*)

How could we get 0, 1, 3, 9, 27, .....?

Solution:

- We know that  $\frac{1}{1-3x} = 1 + 3x + 9x^2 + 27x^3 + \dots$
- Start with the sequence and *shift* it over by 1.
- But how do you do this?
- To see how **shifting** works,

---

67

## Example (24) : Multiplying by x (*shift*)

$$\frac{1}{1-3x} = 1 + 3x + 9x^2 + 27x^3 + \dots \text{ which generates } 1, 3, 9, 27, \dots$$

- To get the zero out front, we need the generating series to look like  $x + 3x^2 + 9x^3 + 27x^4 + \dots$  (**so there is no constant term**).
- **Multiplying by x** has this effect. So the generating function for 0, 1, 3, 9, 27, .... is  $\frac{x}{1-3x}$ .

---

68

## Example (25) : substitution

If we **replace**  $x$  in our **original** generating function by  $x^2$

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

we get

$$\frac{1}{1-x^2} = 1 + x^2 + x^4 + x^6 + \dots \text{ which generates } 1, 0, 1, 0, 1, 0, \dots$$

---

69

## Example (26) : Multiplying by $x$ (*shift*)

How could we get  $0, 1, 0, 1, 0, 1, \dots$ ?

**Solution:**

We know that

$$\frac{1}{1-x^2} = 1 + x^2 + x^4 + x^6 + \dots \text{ which generates } 1, 0, 1, 0, 1, 0, \dots$$

- **Multiplying by  $x$ .** So the generating function for  $0, 1, 0, 1, 0, 1, \dots$  is  $\frac{x}{1-x^2}$

$$\frac{x}{1-x^2} = x + x^3 + x^5 + \dots \text{ which generates } 0, 1, 0, 1, 0, 1, \dots$$

---

70

## Example (27) : addition

What if we add the sequences 1, 0, 1, 0, 1, 0, .... and 0, 1, 0, 1, 0, 1, .... term by term?

Solution:

- We should get 1, 1, 1, 1, 1, 1, .... What happens when we add the generating functions? It works (try it)!

$$\frac{1}{1-x^2} = 1 + x^2 + x^4 + x^6 + \dots \text{ which generates } 1, 0, 1, 0, 1, 0, \dots$$

$$\frac{x}{1-x^2} = x + x^3 + x^5 + \dots \text{ which generates } 0, 1, 0, 1, 0, 1, \dots$$

$$\frac{1}{1-x^2} + \frac{x}{1-x^2} = \frac{1}{1-x}.$$

71

## Example (28) : Differentiation

- what happens if you take the **derivative** of  $\frac{1}{1-x}$ ?

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots \text{ which generates } 1, 1, 1, 1, \dots$$

- We get  $\frac{1}{(1-x)^2}$ .

- On the other hand, if we **differentiate** term by term in the power series, we get

➤  $(1 + x + x^2 + x^3 + \dots)' = 1 + 2x + 3x^2 + 4x^3 + \dots$

➤ which is the generating series for 1, 2, 3, 4, ....

➤ This says

The generating function for 1, 2, 3, 4, 5, ... is  $\frac{1}{(1-x)^2}$

72

## Example (29) : Differentiation

- what happens if you take the **derivative** of  $\frac{1}{1-x}$ ?

I.e. take the **derivative** for **original** generating function

$$\frac{1}{1-x} = \mathbf{1 + x + x^2 + x^3 + \dots} \text{ which generates } 1, 1, 1, 1, \dots$$

- We get

$$\frac{1}{(1-x)^2} = \mathbf{1 + 2x + 3x^2 + 4x^3 + \dots} \text{ which generates } 1, 2, 3, 4, \dots$$

---

73

## Example (30) : Second derivative

- Take a **second derivative**:

$$\frac{1}{(1-x)^2} = \mathbf{1 + 2x + 3x^2 + 4x^3 + \dots} \text{ which generates } 1, 2, 3, 4, \dots$$

$$\frac{2}{(1-x)^3} = 2 + 6x + 12x^2 + 20x^3 + \dots$$

- So  $\frac{1}{(1-x)^3} = 1 + 3x + 6x^2 + 10x^3 + \dots$  is a generating function for the sequence 1, 3, 6, 10, ....

$$\frac{1}{(1-x)^3} = \mathbf{1 + 3x + 6x^2 + 10x^3 + \dots} \text{ which generates } 1, 3, 6, 10, \dots$$

---

74

## Assignments

### Useful Facts About Power Series

#### Exercise 1.

The function  $f(x) = \frac{1}{1-x}$  is the generating function of the sequence 1, 1, 1, ..., because

$$\sum_{k=0}^{\infty} x^k = 1 + x + x^2 + \dots = \frac{1}{1-x} \text{ when } |x| < 1.$$

#### Exercise 2.

The function  $f(x) = \frac{1}{1-ax}$  is the generating function of the sequence 1,  $a$ ,  $a^2$ , ..., because

$$\sum_{k=0}^{\infty} (ax)^k = 1 + ax + a^2x^2 + \dots = \frac{1}{1-ax} \text{ when } |ax| < 1 \text{ for } a \neq 0.$$

### Exercise 3.

Find the generating functions for the sequences

$\{a_k\}$  with

$$(1) \ a_k = 3$$

$$(3) \ a_k = 2^k$$

$$(1) \ G(x) = \sum_{k=0}^{\infty} a_k x^k = \dots, \dots, \dots$$

$$(3) \ G(x) = \sum_{k=0}^{\infty} a_k x^k = \dots, \dots, \dots$$

---

77

### Example (\*)

What is the generating function for the sequence 1,1,1,1,1,1 ?

**Sol :**

$$a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5$$

$$G(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots$$

$$= 1 + x + x^2 + \dots + x^5 \quad (\text{expansion})$$

$$= \frac{x^6 - 1}{x - 1} \quad (\text{closed form})$$

---

78

## Theorem

### Theorem

Let  $f(x) = \sum_{k=0}^{\infty} a_k x^k$  and  $g(x) = \sum_{k=0}^{\infty} b_k x^k$ .

Then  $f(x) + g(x) = \sum_{k=0}^{\infty} (a_k + b_k) x^k$ .

---

79

## Section No. 08

Answer the following Questions.

1.	<p>Find <math>f(1)</math>, <math>f(2)</math>, <math>f(3)</math>, and <math>f(4)</math> if <math>f(n)</math> is defined recursively by <math>f(0) = 1</math> and for <math>n = 0, 1, 2, \dots</math></p> <p><b>a)</b> <math>f(n + 1) = f(n) + 2</math>.  <b>b)</b> <math>f(n + 1) = 3f(n)</math>.  <b>c)</b> <math>f(n + 1) = 2^{f(n)}</math>.  <b>d)</b> <math>f(n + 1) = f(n)^2 + f(n) + 1</math>.</p>
2.	<p>Give a recursive definition of the sequence <math>\{a_n\}</math>, <math>n = 1, 2, 3, \dots</math> if</p> <p><b>a)</b> <math>a_n = 6n</math>.                                   <b>b)</b> <math>a_n = 2n + 1</math>.  <b>c)</b> <math>a_n = 10^n</math>.                                   <b>d)</b> <math>a_n = 5</math>.</p>
3.	<p>Starting with the generating function for <math>1, 2, 3, 4, \dots, 1, 2, 3, 4, \dots</math>, find a generating function for each of the following sequences.</p> <p>a. <math>1, 0, 2, 0, 3, 0, 4, \dots</math>  b. <math>1, -2, 3, -4, 5, -6, \dots</math>  c. <math>0, 3, 6, 9, 12, 15, 18, \dots</math>  d. <math>0, 3, 9, 18, 30, 45, 63, \dots</math> (Hint: relate this sequence to the previous one.)</p> <p><b>The answer</b></p> <p>a. <math>\frac{1}{(1-x^2)^2}</math>.  b. <math>\frac{1}{(1+x)^2}</math>.  c. <math>\frac{3x}{(1-x)^3}</math>.  d. <math>\frac{3x}{(1-x)^3}</math>. (partial sums).</p>
4.	<p>Find the generating function for the sequence <math>1, -2, 4, -8, 16, \dots</math></p> <p><b>The answer</b></p> <p><math>\frac{1}{1+2x}</math>.</p>
5.	<p>Find the generating function for the sequence <math>1, 1, 1, 2, 3, 4, 5, 6, \dots</math></p> <p><b>The answer</b></p> <p><math>\frac{x^3}{(1-x)^2} + \frac{1}{1-x}</math>.</p>

6. Find the generating function for each of the following sequences by relating them back to a sequence with known generating function.
- 4, 4, 4, 4, ...
  - 2, 4, 6, 8, 10, ...
  - 0, 0, 0, 2, 4, 6, 8, 10, ...
  - 1, 5, 25, 125, ...
  - 1, -3, 9, -27, 81, ...
  - 1, 0, 5, 0, 25, 0, 125, 0, ...
  - 0, 1, 0, 0, 2, 0, 0, 3, 0, 0, 4, 0, 0, 5, ...

**The answer**

- $\frac{4}{1-x}$ .
- $\frac{2}{(1-x)^2}$ .
- $\frac{2x^3}{(1-x)^2}$ .
- $\frac{1}{1-5x}$ .
- $\frac{1}{1+3x}$ .
- $\frac{1}{1-5x^2}$ .
- $\frac{x}{(1-x^3)^2}$ .

7. Explain how we know that  $\frac{1}{(1-x)^2}$  is the generating function for  
1, 2, 3, 4, ...

**The answer**

Starting with  $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$ , we can take derivatives of both sides, given  $\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + \dots$ . By the definition of generating functions, this says that  $\frac{1}{(1-x)^2}$  generates the sequence 1, 2, 3, ... You can also find this using differencing or by multiplying.

8. Find the sequence generated by the following generating functions:

a.  $\frac{4x}{1-x}$ .

b.  $\frac{1}{1-4x}$ .

c.  $\frac{x}{1+x}$ .

d.  $\frac{3x}{(1+x)^2}$ .

e.  $\frac{1+x+x^2}{(1-x)^2}$  (Hint: multiplication).

The answer

a. 0, 4, 4, 4, 4, 4, ...

b. 1, 4, 16, 64, 256, ...

c. 0, 1, -1, 1, -1, 1, -1, ...

d. 0, 3, -6, 9, -12, 15, -18, ...

e. 1, 3, 6, 9, 12, 15, ...

# **Chapter (8)**

## **Modeling Computations**

# 13

# Modeling Computation

**13.1** Languages and Grammars

**13.2** Finite-State Machines with Output

**13.3** Finite-State Machines with No Output

**13.4** Language Recognition

**13.5** Turing Machines

**C**omputers can perform many tasks. Given a task, two questions arise. The first is: Can it be carried out using a computer? Once we know that this first question has an affirmative answer, we can ask the second question: How can the task be carried out? Models of computation are used to help answer these questions.

We will study three types of structures used in models of computation, namely, grammars, finite-state machines, and Turing machines. Grammars are used to generate the words of a language and to determine whether a word is in a language. Formal languages, which are generated by grammars, provide models both for natural languages, such as English, and for programming languages, such as Pascal, Fortran, Prolog, C, and Java. In particular, grammars are extremely important in the construction and theory of compilers. The grammars that we will discuss were first used by the American linguist Noam Chomsky in the 1950s.

Various types of finite-state machines are used in modeling. All finite-state machines have a set of states, including a starting state, an input alphabet, and a transition function that assigns a next state to every pair of a state and an input. The states of a finite-state machine give it limited memory capabilities. Some finite-state machines produce an output symbol for each transition; these machines can be used to model many kinds of machines, including vending machines, delay machines, binary adders, and language recognizers. We will also study finite-state machines that have no output but do have final states. Such machines are extensively used in language recognition. The strings that are recognized are those that take the starting state to a final state. The concepts of grammars and finite-state machines can be tied together. We will characterize those sets that are recognized by a finite-state machine and show that these are precisely the sets that are generated by a certain type of grammar.

Finally, we will introduce the concept of a Turing machine. We will show how Turing machines can be used to recognize sets. We will also show how Turing machines can be used to compute number-theoretic functions. We will discuss the Church-Turing thesis, which states that every effective computation can be carried out using a Turing machine. We will explain how Turing machines can be used to study the difficulty of solving certain classes of problems. In particular, we will describe how Turing machines are used to classify problems as tractable versus intractable and solvable versus unsolvable.

## 13.1 Languages and Grammars

### Introduction

Words in the English language can be combined in various ways. The grammar of English tells us whether a combination of words is a valid sentence. For instance, *the frog writes neatly* is a valid sentence, because it is formed from a noun phrase, *the frog*, made up of the article *the* and the noun *frog*, followed by a verb phrase, *writes neatly*, made up of the verb *writes* and the adverb *neatly*. We do not care that this is a nonsensical statement, because we are concerned only with the **syntax**, or form, of the sentence, and not its **semantics**, or meaning. We also note that the combination of words *swims quickly mathematics* is not a valid sentence because it does not follow the rules of English grammar.

The syntax of a **natural language**, that is, a spoken language, such as English, French, German, or Spanish, is extremely complicated. In fact, it does not seem possible to specify all the rules of syntax for a natural language. Research in the automatic translation of one language

to another has led to the concept of a **formal language**, which, unlike a natural language, is specified by a well-defined set of rules of syntax. Rules of syntax are important not only in linguistics, the study of natural languages, but also in the study of programming languages.

We will describe the sentences of a formal language using a grammar. The use of grammars helps when we consider the two classes of problems that arise most frequently in applications to programming languages: (1) How can we determine whether a combination of words is a valid sentence in a formal language? (2) How can we generate the valid sentences of a formal language? Before giving a technical definition of a grammar, we will describe an example of a grammar that generates a subset of English. This subset of English is defined using a list of rules that describe how a valid sentence can be produced. We specify that

1. a **sentence** is made up of a **noun phrase** followed by a **verb phrase**;
2. a **noun phrase** is made up of an **article** followed by an **adjective** followed by a **noun**, or
3. a **noun phrase** is made up of an **article** followed by a **noun**;
4. a **verb phrase** is made up of a **verb** followed by an **adverb**, or
5. a **verb phrase** is made up of a **verb**;
6. an **article** is *a*, or
7. an **article** is *the*;
8. an **adjective** is *large*, or
9. an **adjective** is *hungry*;
10. a **noun** is *rabbit*, or
11. a **noun** is *mathematician*;
12. a **verb** is *eats*, or
13. a **verb** is *hops*;
14. an **adverb** is *quickly*, or
15. an **adverb** is *wildly*.

From these rules we can form valid sentences using a series of replacements until no more rules can be used. For instance, we can follow the sequence of replacements:

```

sentence
noun phrase verb phrase
article adjective noun verb phrase
article adjective noun verb adverb
the adjective noun verb adverb
the large noun verb adverb
the large rabbit verb adverb
the large rabbit hops adverb
the large rabbit hops quickly
```

to obtain a valid sentence. It is also easy to see that some other valid sentences are: *a hungry mathematician eats wildly*, *a large mathematician hops*, *the rabbit eats quickly*, and so on. Also, we can see that *the quickly eats mathematician* is not a valid sentence.

## **Phrase-Structure Grammars**

Before we give a formal definition of a grammar, we introduce a little terminology.

**DEFINITION 1**

A *vocabulary* (or *alphabet*)  $V$  is a finite, nonempty set of elements called *symbols*. A *word* (or *sentence*) over  $V$  is a string of finite length of elements of  $V$ . The *empty string* or *null string*, denoted by  $\lambda$ , is the string containing no symbols. The set of all words over  $V$  is denoted by  $V^*$ . A *language* over  $V$  is a subset of  $V^*$ .

Note that  $\lambda$ , the empty string, is the string containing no symbols. It is different from  $\emptyset$ , the empty set. It follows that  $\{\lambda\}$  is the set containing exactly one string, namely, the empty string.

Languages can be specified in various ways. One way is to list all the words in the language. Another is to give some criteria that a word must satisfy to be in the language. In this section, we describe another important way to specify a language, namely, through the use of a grammar, such as the set of rules we gave in the introduction to this section. A grammar provides a set of symbols of various types and a set of rules for producing words. More precisely, a grammar has a **vocabulary**  $V$ , which is a set of symbols used to derive members of the language. Some of the elements of the vocabulary cannot be replaced by other symbols. These are called **terminals**, and the other members of the vocabulary, which can be replaced by other symbols, are called **nonterminals**. The sets of terminals and nonterminals are usually denoted by  $T$  and  $N$ , respectively. In the example given in the introduction of the section, the set of terminals is  $\{a, \text{the}, \text{rabbit}, \text{mathematician}, \text{hops}, \text{eats}, \text{quickly}, \text{wildly}\}$ , and the set of nonterminals is  $\{\text{sentence, noun phrase, verb phrase, adjective, article, noun, verb, adverb}\}$ . There is a special member of the vocabulary called the **start symbol**, denoted by  $S$ , which is the element of the vocabulary that we always begin with. In the example in the introduction, the start symbol is **sentence**. The rules that specify when we can replace a string from  $V^*$ , the set of all strings of elements in the vocabulary, with another string are called the **productions** of the grammar. We denote by  $z_0 \rightarrow z_1$  the production that specifies that  $z_0$  can be replaced by  $z_1$  within a string. The productions in the grammar given in the introduction of this section were listed. The first production, written using this notation, is **sentence  $\rightarrow$  noun phrase verb phrase**. We summarize this terminology in Definition 2.

The notion of a phrase-structure grammar extends the concept of a rewrite system devised by Axel Thue in the early 20th century.

**DEFINITION 2**

A *phrase-structure grammar*  $G = (V, T, S, P)$  consists of a vocabulary  $V$ , a subset  $T$  of  $V$  consisting of terminal symbols, a start symbol  $S$  from  $V$ , and a finite set of productions  $P$ . The set  $V - T$  is denoted by  $N$ . Elements of  $N$  are called *nonterminal symbols*. Every production in  $P$  must contain at least one nonterminal on its left side.

**EXAMPLE 1**

Let  $G = (V, T, S, P)$ , where  $V = \{a, b, A, B, S\}$ ,  $T = \{a, b\}$ ,  $S$  is the start symbol, and  $P = [S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b]$ .  $G$  is an example of a phrase-structure grammar.  $\blacktriangleleft$

We will be interested in the words that can be generated by the productions of a phrase-structure grammar.

**DEFINITION 3**

Let  $G = (V, T, S, P)$  be a phrase-structure grammar. Let  $w_0 = lz_0r$  (that is, the concatenation of  $l$ ,  $z_0$ , and  $r$ ) and  $w_1 = lz_1r$  be strings over  $V$ . If  $z_0 \rightarrow z_1$  is a production of  $G$ , we say that  $w_1$  is *directly derivable* from  $w_0$  and we write  $w_0 \Rightarrow w_1$ . If  $w_0, w_1, \dots, w_n$  are strings over  $V$  such that  $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \dots, w_{n-1} \Rightarrow w_n$ , then we say that  $w_n$  is *derivable* from  $w_0$ , and we write  $w_0 \xrightarrow{*} w_n$ . The sequence of steps used to obtain  $w_n$  from  $w_0$  is called a *derivation*.

**EXAMPLE 2** The string  $Aaba$  is directly derivable from  $ABA$  in the grammar in Example 1 because  $B \rightarrow ab$  is a production in the grammar. The string  $abababa$  is derivable from  $ABA$  because  $ABA \Rightarrow Aaba \Rightarrow BBaba \Rightarrow Bababa \Rightarrow abababa$ , using the productions  $B \rightarrow ab$ ,  $A \rightarrow BB$ ,  $B \rightarrow ab$ , and  $B \rightarrow ab$  in succession. 

#### DEFINITION 4

Let  $G = (V, T, S, P)$  be a phrase-structure grammar. The *language generated by G* (or the *language of G*), denoted by  $L(G)$ , is the set of all strings of terminals that are derivable from the starting state  $S$ . In other words,

$$L(G) = \{w \in T^* \mid S \xrightarrow{*} w\}.$$

In Examples 3 and 4 we find the language generated by a phrase-structure grammar.

**EXAMPLE 3** Let  $G$  be the grammar with vocabulary  $V = \{S, A, a, b\}$ , set of terminals  $T = \{a, b\}$ , starting symbol  $S$ , and productions  $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$ . What is  $L(G)$ , the language of this grammar?

**Solution:** From the start state  $S$  we can derive  $aA$  using the production  $S \rightarrow aA$ . We can also use the production  $S \rightarrow b$  to derive  $b$ . From  $aA$  the production  $A \rightarrow aa$  can be used to derive  $aaa$ . No additional words can be derived. Hence,  $L(G) = \{b, aaa\}$ . 

**EXAMPLE 4** Let  $G$  be the grammar with vocabulary  $V = \{S, 0, 1\}$ , set of terminals  $T = \{0, 1\}$ , starting symbol  $S$ , and productions  $P = \{S \rightarrow 11S, S \rightarrow 0\}$ . What is  $L(G)$ , the language of this grammar?

**Solution:** From  $S$  we can derive 0 using  $S \rightarrow 0$ , or  $11S$  using  $S \rightarrow 11S$ . From  $11S$  we can derive either  $110$  or  $1111S$ . From  $1111S$  we can derive  $11110$  and  $111111S$ . At any stage of a derivation we can either add two 1s at the end of the string or terminate the derivation by adding a 0 at the end of the string. We surmise that  $L(G) = \{0, 110, 11110, 1111110, \dots\}$ , the set of all strings that begin with an even number of 1s and end with a 0. This can be proved using an inductive argument that shows that after  $n$  productions have been used, the only strings of terminals generated are those consisting of  $n - 1$  concatenations of 11 followed by 0. (This is left as an exercise for the reader.) 

The problem of constructing a grammar that generates a given language often arises. Examples 5, 6, and 7 describe problems of this kind.

**EXAMPLE 5** Give a phrase-structure grammar that generates the set  $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$ .

**Solution:** Two productions can be used to generate all strings consisting of a string of 0s followed by a string of the same number of 1s, including the null string. The first builds up successively longer strings in the language by adding a 0 at the start of the string and a 1 at the end. The second production replaces  $S$  with the empty string. The solution is the grammar  $G = (V, T, S, P)$ , where  $V = \{0, 1, S\}$ ,  $T = \{0, 1\}$ ,  $S$  is the starting symbol, and the productions are

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow \lambda. \end{aligned}$$

The verification that this grammar generates the correct set is left as an exercise for the reader. 

Example 5 involved the set of strings made up of 0s followed by 1s, where the number of 0s and 1s are the same. Example 6 considers the set of strings consisting of 0s followed by 1s, where the number of 0s and 1s may differ.

**EXAMPLE 6** Find a phrase-structure grammar to generate the set  $\{0^m 1^n \mid m \text{ and } n \text{ are nonnegative integers}\}$ .

**Solution:** We will give two grammars  $G_1$  and  $G_2$  that generate this set. This will illustrate that two grammars can generate the same language.

The grammar  $G_1$  has alphabet  $V = \{S, 0, 1\}$ ; terminals  $T = \{0, 1\}$ ; and productions  $S \rightarrow 0S$ ,  $S \rightarrow S1$ , and  $S \rightarrow \lambda$ .  $G_1$  generates the correct set, because using the first production  $m$  times puts  $m$  0s at the beginning of the string, and using the second production  $n$  times puts  $n$  1s at the end of the string. The details of this verification are left to the reader.

The grammar  $G_2$  has alphabet  $V = \{S, A, 0, 1\}$ ; terminals  $T = \{0, 1\}$ ; and productions  $S \rightarrow 0S$ ,  $S \rightarrow 1A$ ,  $S \rightarrow 1$ ,  $A \rightarrow 1A$ ,  $A \rightarrow 1$ , and  $S \rightarrow \lambda$ . The details that this grammar generates the correct set are left as an exercise for the reader. 

Sometimes a set that is easy to describe can be generated only by a complicated grammar. Example 7 illustrates this.

**EXAMPLE 7** One grammar that generates the set  $\{0^n 1^n 2^n \mid n = 0, 1, 2, 3, \dots\}$  is  $G = (V, T, S, P)$  with  $V = \{0, 1, 2, S, A, B, C\}$ ;  $T = \{0, 1, 2\}$ ; starting state  $S$ ; and productions  $S \rightarrow C$ ,  $C \rightarrow 0CAB$ ,  $S \rightarrow \lambda$ ,  $BA \rightarrow AB$ ,  $0A \rightarrow 01$ ,  $1A \rightarrow 11$ ,  $1B \rightarrow 12$ , and  $2B \rightarrow 22$ . We leave it as an exercise for the reader (Exercise 12) to show that this statement is correct. The grammar given is the simplest type of grammar that generates this set, in a sense that will be made clear later in this section. 

## Types of Phrase-Structure Grammars



Phrase-structure grammars can be classified according to the types of productions that are allowed. We will describe the classification scheme introduced by Noam Chomsky. In Section 13.4 we will see that the different types of languages defined in this scheme correspond to the classes of languages that can be recognized using different models of computing machines.

A **type 0** grammar has no restrictions on its productions. A **type 1** grammar can have productions of the form  $w_1 \rightarrow w_2$ , where  $w_1 = lAr$  and  $w_2 = lwr$ , where  $A$  is a nonterminal symbol,  $l$  and  $r$  are strings of zero or more terminal or nonterminal symbols, and  $w$  is a nonempty string of terminal or nonterminal symbols. It can also have the production  $S \rightarrow \lambda$  as long as  $S$  does not appear on the right-hand side of any other production. A **type 2** grammar can have productions only of the form  $w_1 \rightarrow w_2$ , where  $w_1$  is a single symbol that is not a terminal symbol. A **type 3** grammar can have productions only of the form  $w_1 \rightarrow w_2$  with  $w_1 = A$  and either  $w_2 = aB$  or  $w_2 = a$ , where  $A$  and  $B$  are nonterminal symbols and  $a$  is a terminal symbol, or with  $w_1 = S$  and  $w_2 = \lambda$ .

Type 2 grammars are called **context-free grammars** because a nonterminal symbol that is the left side of a production can be replaced in a string whenever it occurs, no matter what else is in the string. A language generated by a type 2 grammar is called a **context-free language**. When there is a production of the form  $lw_1r \rightarrow lw_2r$  (but not of the form  $w_1 \rightarrow w_2$ ), the grammar is called **type 1** or **context-sensitive** because  $w_1$  can be replaced by  $w_2$  only when it is surrounded by the strings  $l$  and  $r$ . A language generated by a type 1 grammar is called a **context-sensitive language**. Type 3 grammars are also called **regular grammars**. A language generated by a regular grammar is called **regular**. Section 13.4 deals with the relationship between regular languages and finite-state machines.

Of the four types of grammars we have defined, context-sensitive grammars have the most complicated definition. Sometimes, these grammars are defined in a different way. A production of the form  $w_1 \rightarrow w_2$  is called **noncontracting** if the length of  $w_1$  is less than or equal to the

length of  $w_2$ . According to our characterization of context-sensitive languages, every production in a type 1 grammar, other than the production  $S \rightarrow \lambda$ , if it is present, is noncontracting. It follows that the lengths of the strings in a derivation in a context-sensitive language are nondecreasing unless the production  $S \rightarrow \lambda$  is used. This means that the only way for the empty string to belong to the language generated by a context-sensitive grammar is for the production  $S \rightarrow \lambda$  to be part of the grammar. The other way that context-sensitive grammars are defined is by specifying that all productions are noncontracting. A grammar with this property is called **noncontracting** or **monotonic**. The class of noncontracting grammars is not the same as the class of context-sensitive grammars. However, these two classes are closely related; it can be shown that they define the same set of languages except that noncontracting grammars cannot generate any language containing the empty string  $\lambda$ .

**EXAMPLE 8**

From Example 6 we know that  $\{0^m 1^n \mid m, n = 0, 1, 2, \dots\}$  is a regular language, because it can be generated by a regular grammar, namely, the grammar  $G_2$  in Example 6.

Context-free and regular grammars play an important role in programming languages. Context-free grammars are used to define the syntax of almost all programming languages. These grammars are strong enough to define a wide range of languages. Furthermore, efficient algorithms can be devised to determine whether and how a string can be generated. Regular grammars are used to search text for certain patterns and in lexical analysis, which is the process of transforming an input stream into a stream of tokens for use by a parser.

**EXAMPLE 9**

It follows from Example 5 that  $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$  is a context-free language, because the productions in this grammar are  $S \rightarrow 0S1$  and  $S \rightarrow \lambda$ . However, it is not a regular language. This will be shown in Section 13.4.

**EXAMPLE 10**

The set  $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$  is a context-sensitive language, because it can be generated by a type 1 grammar, as Example 7 shows, but not by any type 2 language. (This is shown in Exercise 28 in the supplementary exercises at the end of the chapter.)

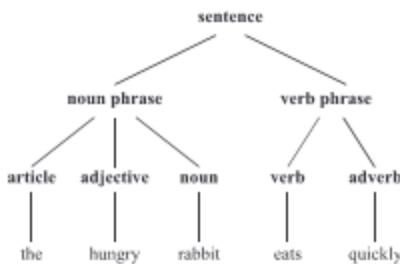
Table 1 summarizes the terminology used to classify phrase-structure grammars.

## Derivation Trees

A derivation in the language generated by a context-free grammar can be represented graphically using an ordered rooted tree, called a **derivation**, or **parse tree**. The root of this tree represents the starting symbol. The internal vertices of the tree represent the nonterminal symbols that arise in the derivation. The leaves of the tree represent the terminal symbols that arise. If the production  $A \rightarrow w$  arises in the derivation, where  $w$  is a word, the vertex that represents  $A$  has as children vertices that represent each symbol in  $w$ , in order from left to right.

**TABLE 1** Types of Grammars.

Type	Restrictions on Productions $w_1 \rightarrow w_2$
0	No restrictions
1	$w_1 = lAr$ and $w_2 = lwr$ , where $A \in N$ , $l, r, w \in (N \cup T)^*$ and $w \neq \lambda$ ; or $w_1 = S$ and $w_2 = \lambda$ as long as $S$ is not on the right-hand side of another production
2	$w_1 = A$ , where $A$ is a nonterminal symbol
3	$w_1 = A$ and $w_2 = aB$ or $w_2 = a$ , where $A \in N$ , $B \in N$ , and $a \in T$ ; or $w_1 = S$ and $w_2 = \lambda$

**FIGURE 1 A Derivation Tree.**

**EXAMPLE 11** Construct a derivation tree for the derivation of *the hungry rabbit eats quickly*, given in the introduction of this section.

*Solution:* The derivation tree is shown in Figure 1. 

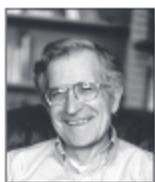
The problem of determining whether a string is in the language generated by a context-free grammar arises in many applications, such as in the construction of compilers. Two approaches to this problem are indicated in Example 12.

**EXAMPLE 12** Determine whether the word *cbaB* belongs to the language generated by the grammar  $G = (V, T, S, P)$ , where  $V = \{a, b, c, A, B, C, S\}$ ,  $T = \{a, b, c\}$ ,  $S$  is the starting symbol, and the productions are

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow Ca \\ B &\rightarrow Ba \\ B &\rightarrow Cb \\ B &\rightarrow b \\ C &\rightarrow cb \\ C &\rightarrow b. \end{aligned}$$

*Solution:* One way to approach this problem is to begin with  $S$  and attempt to derive *cbaB* using a series of productions. Because there is only one production with  $S$  on its left-hand side, we must start with  $S \rightarrow AB$ . Next we use the only production that has  $A$  on its left-hand side, namely,  $A \rightarrow Ca$ , to obtain  $S \rightarrow AB \rightarrow CaB$ . Because *cbaB* begins with the symbols *cb*, we use the production  $C \rightarrow cb$ . This gives us  $S \rightarrow AB \rightarrow CaB \rightarrow cbaB$ . We finish by using the production  $B \rightarrow b$ , to obtain  $S \rightarrow AB \rightarrow CaB \rightarrow cbaB \rightarrow cbaB$ . The approach that we have used is called **top-down parsing**, because it begins with the starting symbol and proceeds by successively applying productions.

There is another approach to this problem, called **bottom-up parsing**. In this approach, we work backward. Because *cbaB* is the string to be derived, we can use the production  $C \rightarrow cb$ , so



**AVRAM NOAM CHOMSKY (BORN 1928)** Noam Chomsky, born in Philadelphia, is the son of a Hebrew scholar. He received his B.A., M.A., and Ph.D. in linguistics, all from the University of Pennsylvania. He was on the staff of the University of Pennsylvania from 1950 until 1951. In 1955 he joined the faculty at M.I.T., beginning his M.I.T. career teaching engineers French and German. Chomsky is currently the Ferrari P. Ward Professor of foreign languages and linguistics at M.I.T. He is known for his many fundamental contributions to linguistics, including the study of grammars. Chomsky is also widely known for his outspoken political activism.

that  $Cab \rightarrow cbab$ . Then, we can use the production  $A \rightarrow Ca$ , so that  $Ab \rightarrow Cab \rightarrow cbab$ . Using the production  $B \rightarrow b$  gives  $AB \rightarrow Ab \rightarrow Cab \rightarrow cbab$ . Finally, using  $S \rightarrow AB$  shows that a complete derivation for  $cbab$  is  $S \rightarrow AB \rightarrow Ab \rightarrow Cab \rightarrow cbab$ .  $\blacktriangleleft$

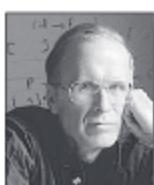
## Backus–Naur Form



The ancient Indian grammarian Pāṇini specified Sanskrit using 3959 rules; Backus–Naur form is sometimes called Backus–Pāṇini form.

There is another notation that is sometimes used to specify a type 2 grammar, called the **Backus–Naur form (BNF)**, after John Backus, who invented it, and Peter Naur, who refined it for use in the specification of the programming language ALGOL. (Surprisingly, a notation quite similar to the Backus–Naur form was used approximately 2500 years ago to describe the grammar of Sanskrit.) The Backus–Naur form is used to specify the syntactic rules of many computer languages, including Java. The productions in a type 2 grammar have a single nonterminal symbol as their left-hand side. Instead of listing all the productions separately, we can combine all those with the same nonterminal symbol on the left-hand side into one statement. Instead of using the symbol  $\rightarrow$  in a production, we use the symbol  $::=$ . We enclose all nonterminal symbols in brackets,  $\langle \rangle$ , and we list all the right-hand sides of productions in the same statement, separating them by bars. For instance, the productions  $A \rightarrow Aa$ ,  $A \rightarrow a$ , and  $A \rightarrow AB$  can be combined into  $\langle A \rangle ::= \langle A \rangle a \mid a \mid \langle A \rangle \langle B \rangle$ .

Example 13 illustrates how the Backus–Naur form is used to describe the syntax of programming languages. Our example comes from the original use of Backus–Naur form in the description of ALGOL 60.



**JOHN BACKUS (1924–2007)** John Backus was born in Philadelphia and grew up in Wilmington, Delaware. He attended the Hill School in Pottstown, Pennsylvania. He needed to attend summer school every year because he disliked studying and was not a serious student. But he enjoyed spending his summers in New Hampshire where he attended summer school and amused himself with summer activities, including sailing. He obliged his father by enrolling at the University of Virginia to study chemistry. But he quickly decided chemistry was not for him, and in 1943 he entered the army, where he received medical training and worked in a neurosurgery ward in an army hospital. Ironically, Backus was soon diagnosed with a bone tumor in his skull and was fitted with a metal plate. His medical work in the army convinced him to try medical school, but he abandoned this after nine months because he disliked the rote memorization required. After dropping out of medical school, he entered a school for radio technicians because he wanted to build his own high fidelity set. A teacher in this school recognized his potential and asked him to help with some mathematical calculations needed for an article in a magazine. Finally, Backus found what he was interested in: mathematics and its applications. He enrolled at Columbia University, from which he received both bachelor's and master's degrees in mathematics. Backus joined IBM as a programmer in 1950. He participated in the design and development of two of IBM's early computers. From 1954 to 1958 he led the IBM group that developed FORTRAN. Backus became a staff member at the IBM Watson Research Center in 1958. He was part of the committees that designed the programming language ALGOL, using what is now called the Backus–Naur form for the description of the syntax of this language. Later, Backus worked on the mathematics of families of sets and on a functional style of programming. Backus became an IBM Fellow in 1963, and he received the National Medal of Science in 1974 and the prestigious Turing Award from the Association of Computing Machinery in 1977.



**PETER NAUR (BORN 1928)** Peter Naur was born in Frederiksberg, near Copenhagen. As a boy he became interested in astronomy. Not only did he observe heavenly bodies, but he also computed the orbits of comets and asteroids. Naur attended Copenhagen University, receiving his degree in 1949. He spent 1950 and 1951 in Cambridge, where he used an early computer to calculate the motions of comets and planets. After returning to Denmark he continued working in astronomy but kept his ties to computing. In 1955 he served as a consultant to the building of the first Danish computer. In 1959 Naur made the switch from astronomy to computing as a full-time activity. His first job as full-time computer scientist was participating in the development of the programming language ALGOL. From 1960 to 1967 he worked on the development of compilers for ALGOL and COBOL. In 1969 he became professor of computer science at Copenhagen University, where he has worked in the area of programming methodology. His research interests include the design, structure, and performance of computer programs. Naur has been a pioneer in both the areas of software architecture and software engineering. He rejects the view that computer programming is a branch of mathematics and prefers that computer science be called *datalogy*.

**EXAMPLE 13**

In ALGOL 60 an identifier (which is the name of an entity such as a variable) consists of a string of alphanumeric characters (that is, letters and digits) and must begin with a letter. We can use these rules in Backus–Naur to describe the set of allowable identifiers:

$$\begin{aligned}\langle \text{identifier} \rangle &::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle \\ \langle \text{letter} \rangle &::= a \mid b \mid \dots \mid y \mid z \quad \text{the ellipsis indicates that all 26 letters are included} \\ \langle \text{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

For example, we can produce the valid identifier  $x99a$  by using the first rule to replace  $\langle \text{identifier} \rangle$  by  $\langle \text{identifier} \rangle \langle \text{letter} \rangle$ , the second rule to obtain  $\langle \text{identifier} \rangle a$ , the first rule twice to obtain  $\langle \text{identifier} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle a$ , the third rule twice to obtain  $\langle \text{identifier} \rangle 99a$ , the first rule to obtain  $\langle \text{letter} \rangle 99a$ , and finally the second rule to obtain  $x99a$ .

**EXAMPLE 14**

What is the Backus–Naur form of the grammar for the subset of English described in the introduction to this section?

*Solution:* The Backus–Naur form of this grammar is

$$\begin{aligned}\langle \text{sentence} \rangle &::= \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle \\ \langle \text{noun phrase} \rangle &::= \langle \text{article} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle \mid \langle \text{article} \rangle \langle \text{noun} \rangle \\ \langle \text{verb phrase} \rangle &::= \langle \text{verb} \rangle \langle \text{adverb} \rangle \mid \langle \text{verb} \rangle \\ \langle \text{article} \rangle &::= a \mid \text{the} \\ \langle \text{adjective} \rangle &::= \text{large} \mid \text{hungry} \\ \langle \text{noun} \rangle &::= \text{rabbit} \mid \text{mathematician} \\ \langle \text{verb} \rangle &::= \text{eats} \mid \text{hops} \\ \langle \text{adverb} \rangle &::= \text{quickly} \mid \text{wildly}\end{aligned}$$
**EXAMPLE 15**

Give the Backus–Naur form for the production of signed integers in decimal notation. (A **signed integer** is a nonnegative integer preceded by a plus sign or a minus sign.)

*Solution:* The Backus–Naur form for a grammar that produces signed integers is

$$\begin{aligned}\langle \text{signed integer} \rangle &::= \langle \text{sign} \rangle \langle \text{integer} \rangle \\ \langle \text{sign} \rangle &::= + \mid - \\ \langle \text{integer} \rangle &::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle \\ \langle \text{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

The Backus–Naur form, with a variety of extensions, is used extensively to specify the syntax of programming languages, such as Java and LISP; database languages, such as SQL; and markup languages, such as XML. Some extensions of the Backus–Naur form that are commonly used in the description of programming languages are introduced in the preamble to Exercise 34.

## Exercises

Exercises 1–3 refer to the grammar with start symbol **sentence**, set of terminals  $T = \{\text{the}, \text{sleepy}, \text{happy}, \text{tortoise}, \text{hare}, \text{passes}, \text{runs}, \text{quickly}, \text{slowly}\}$ , set of nonterminals  $N = \{\text{noun phrase}, \text{transitive verb phrase}, \text{intransitive verb phrase}, \text{article}, \text{adjective}, \text{noun}, \text{verb}, \text{adverb}\}$ , and productions:

**sentence** → **noun phrase**    **transitive verb phrase**  
**noun phrase**

**sentence** → **noun phrase**    **intransitive verb phrase**  
**noun phrase** → **article**    **adjective**    **noun**  
**noun phrase** → **article**    **noun**  
**transitive verb phrase** → **transitive verb**  
**intransitive verb phrase** → **intransitive verb**    **adverb**  
**intransitive verb phrase** → **intransitive verb**  
**article** → **the**

**adjective** → *sleepy*

**adjective** → *happy*

**noun** → *tortoise*

**noun** → *hare*

**transitive verb** → *passes*

**intransitive verb** → *runs*

**adverb** → *quickly*

**adverb** → *slowly*

1. Use the set of productions to show that each of these sentences is a valid sentence.
  - a) *the happy hare runs*
  - b) *the sleepy tortoise runs quickly*
  - c) *the tortoise passes the hare*
  - d) *the sleepy hare passes the happy tortoise*
2. Find five other valid sentences, besides those given in Exercise 1.
3. Show that *the hare runs the sleepy tortoise* is not a valid sentence.
4. Let  $G = (V, T, S, P)$  be the phrase-structure grammar with  $V = \{0, 1, A, S\}$ ,  $T = \{0, 1\}$ , and set of productions  $P$  consisting of  $S \rightarrow 1S$ ,  $S \rightarrow 00A$ ,  $A \rightarrow 0A$ , and  $A \rightarrow 0$ .
  - a) Show that 111000 belongs to the language generated by  $G$ .
  - b) Show that 11001 does not belong to the language generated by  $G$ .
  - c) What is the language generated by  $G$ ?
5. Let  $G = (V, T, S, P)$  be the phrase-structure grammar with  $V = \{0, 1, A, B, S\}$ ,  $T = \{0, 1\}$ , and set of productions  $P$  consisting of  $S \rightarrow 0A$ ,  $S \rightarrow 1A$ ,  $A \rightarrow 0B$ ,  $B \rightarrow 1A$ ,  $B \rightarrow 1$ .
  - a) Show that 10101 belongs to the language generated by  $G$ .
  - b) Show that 10110 does not belong to the language generated by  $G$ .
  - c) What is the language generated by  $G$ ?
- \*6. Let  $V = \{S, A, B, a, b\}$  and  $T = \{a, b\}$ . Find the language generated by the grammar  $(V, T, S, P)$  when the set  $P$  of productions consists of
  - a)  $S \rightarrow AB$ ,  $A \rightarrow ab$ ,  $B \rightarrow bb$ .
  - b)  $S \rightarrow AB$ ,  $S \rightarrow aA$ ,  $A \rightarrow a$ ,  $B \rightarrow ba$ .
  - c)  $S \rightarrow AB$ ,  $S \rightarrow AA$ ,  $A \rightarrow aB$ ,  $A \rightarrow ab$ ,  $B \rightarrow b$ .
  - d)  $S \rightarrow AA$ ,  $S \rightarrow B$ ,  $A \rightarrow aaA$ ,  $A \rightarrow aa$ ,  $B \rightarrow bb$ ,  $B \rightarrow b$ .
  - e)  $S \rightarrow AB$ ,  $A \rightarrow aAb$ ,  $B \rightarrow bBa$ ,  $A \rightarrow \lambda$ ,  $B \rightarrow \lambda$ .
7. Construct a derivation of  $0^31^3$  using the grammar given in Example 5.
8. Show that the grammar given in Example 5 generates the set  $\{0^n1^n \mid n = 0, 1, 2, \dots\}$ .
  - a) Construct a derivation of  $0^21^4$  using the grammar  $G_1$  in Example 6.
  - b) Construct a derivation of  $0^21^4$  using the grammar  $G_2$  in Example 6.
10. a) Show that the grammar  $G_1$  given in Example 6 generates the set  $\{0^m1^n \mid m, n = 0, 1, 2, \dots\}$ .

b) Show that the grammar  $G_2$  in Example 6 generates the same set.

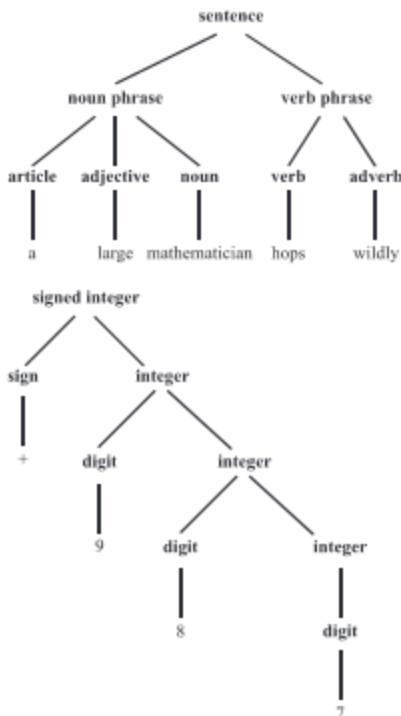
11. Construct a derivation of  $0^21^22^2$  in the grammar given in Example 7.
- \*12. Show that the grammar given in Example 7 generates the set  $\{0^n1^n2^n \mid n = 0, 1, 2, \dots\}$ .
13. Find a phrase-structure grammar for each of these languages.
  - a) the set consisting of the bit strings 0, 1, and 11
  - b) the set of bit strings containing only 1s
  - c) the set of bit strings that start with 0 and end with 1
  - d) the set of bit strings that consist of a 0 followed by an even number of 1s
14. Find a phrase-structure grammar for each of these languages.
  - a) the set consisting of the bit strings 10, 01, and 101
  - b) the set of bit strings that start with 00 and end with one or more 1s
  - c) the set of bit strings consisting of an even number of 1s followed by a final 0
  - d) the set of bit strings that have neither two consecutive 0s nor two consecutive 1s
- \*15. Find a phrase-structure grammar for each of these languages.
  - a) the set of all bit strings containing an even number of 0s and no 1s
  - b) the set of all bit strings made up of a 1 followed by an odd number of 0s
  - c) the set of all bit strings containing an even number of 0s and an even number of 1s
  - d) the set of all strings containing 10 or more 0s and no 1s
  - e) the set of all strings containing more 0s than 1s
  - f) the set of all strings containing an equal number of 0s and 1s
  - g) the set of all strings containing an unequal number of 0s and 1s
16. Construct phrase-structure grammars to generate each of these sets.
 

<b>a)</b> $\{1^n \mid n \geq 0\}$	<b>b)</b> $\{10^n \mid n \geq 0\}$
<b>c)</b> $\{(11)^n \mid n \geq 0\}$	
17. Construct phrase-structure grammars to generate each of these sets.
 

<b>a)</b> $\{0^n \mid n \geq 0\}$	<b>b)</b> $\{1^n0 \mid n \geq 0\}$
<b>c)</b> $\{(000)^n \mid n \geq 0\}$	
18. Construct phrase-structure grammars to generate each of these sets.
 

<b>a)</b> $\{01^{2n} \mid n \geq 0\}$	<b>b)</b> $\{0^n1^{2n} \mid n \geq 0\}$
<b>c)</b> $\{0^n1^m0^n \mid m \geq 0 \text{ and } n \geq 0\}$	
19. Let  $V = \{S, A, B, a, b\}$  and  $T = \{a, b\}$ . Determine whether  $G = (V, T, S, P)$  is a type 0 grammar but not a type 1 grammar, a type 1 grammar but not a type 2 grammar, or a type 2 grammar but not a type 3 grammar if  $P$ , the set of productions, is
  - a)  $S \rightarrow aAB$ ,  $A \rightarrow Bb$ ,  $B \rightarrow \lambda$ .

- b)  $S \rightarrow aA, A \rightarrow a, A \rightarrow b.$   
 c)  $S \rightarrow ABa, AB \rightarrow a.$   
 d)  $S \rightarrow ABA, A \rightarrow ab, B \rightarrow ab.$   
 e)  $S \rightarrow bA, A \rightarrow B, B \rightarrow a.$   
 f)  $S \rightarrow aA, aA \rightarrow B, B \rightarrow aA, A \rightarrow b.$   
 g)  $S \rightarrow bA, A \rightarrow b, S \rightarrow \lambda.$   
 h)  $S \rightarrow AB, B \rightarrow aAb, aAb \rightarrow b.$   
 i)  $S \rightarrow aA, A \rightarrow bB, B \rightarrow b, B \rightarrow \lambda.$   
 j)  $S \rightarrow A, A \rightarrow B, B \rightarrow \lambda.$
20. A **palindrome** is a string that reads the same backward as it does forward, that is, a string  $w$ , where  $w = w^R$ , where  $w^R$  is the reversal of the string  $w$ . Find a context-free grammar that generates the set of all palindromes over the alphabet  $\{0, 1\}$ .
- \*21. Let  $G_1$  and  $G_2$  be context-free grammars, generating the languages  $L(G_1)$  and  $L(G_2)$ , respectively. Show that there is a context-free grammar generating each of these sets.
- a)  $L(G_1) \cup L(G_2)$       b)  $L(G_1)L(G_2)$   
 c)  $L(G_1)^*$
22. Find the strings constructed using the derivation trees shown here.



23. Construct derivation trees for the sentences in Exercise 1.
24. Let  $G$  be the grammar with  $V = \{a, b, c, S\}$ ;  $T = \{a, b, c\}$ ; starting symbol  $S$ ; and productions  $S \rightarrow abS$ ,  $S \rightarrow bcS$ ,  $S \rightarrow bbS$ ,  $S \rightarrow a$ , and  $S \rightarrow cb$ . Construct derivation trees for
- a)  $bcbba.$       b)  $bbbcbba.$   
 c)  $bcabbbbbcbb.$

- \*25. Use top-down parsing to determine whether each of the following strings belongs to the language generated by the grammar in Example 12.
- a)  $baba$       b)  $abab$   
 c)  $caba$       d)  $bbbcba$
- \*26. Use bottom-up parsing to determine whether the strings in Exercise 25 belong to the language generated by the grammar in Example 12.
27. Construct a derivation tree for  $-109$  using the grammar given in Example 15.
28. a) Explain what the productions are in a grammar if the Backus–Naur form for productions is as follows:
- ```

<expression> ::= ((<expression>)) |  

  <expression> + (<expression>) |  

  <expression> * (<expression>) |  

  <variable>  

<variable> ::= x | y
  
```
- b) Find a derivation tree for  $(x * y) + x$  in this grammar.
29. a) Construct a phrase-structure grammar that generates all signed decimal numbers, consisting of a sign, either  $+$  or  $-$ ; a nonnegative integer; and a decimal fraction that is either the empty string or a decimal point followed by a positive integer, where initial zeros in an integer are allowed.  
 b) Give the Backus–Naur form of this grammar.  
 c) Construct a derivation tree for  $-31.4$  in this grammar.
30. a) Construct a phrase-structure grammar for the set of all fractions of the form  $a/b$ , where  $a$  is a signed integer in decimal notation and  $b$  is a positive integer.  
 b) What is the Backus–Naur form for this grammar?  
 c) Construct a derivation tree for  $+311/17$  in this grammar.
31. Give production rules in Backus–Naur form for an identifier if it can consist of
- a) one or more lowercase letters.  
 b) at least three but no more than six lowercase letters.  
 c) one to six uppercase or lowercase letters beginning with an uppercase letter.  
 d) a lowercase letter, followed by a digit or an underscore, followed by three or four alphanumeric characters (lower or uppercase letters and digits).
32. Give production rules in Backus–Naur form for the name of a person if this name consists of a first name, which is a string of letters, where only the first letter is uppercase; a middle initial; and a last name, which can be any string of letters.
33. Give production rules in Backus–Naur form that generate all identifiers in the C programming language. In C an identifier starts with a letter or an underscore ( $_$ ) that is followed by one or more lowercase letters, uppercase letters, underscores, and digits.

 Several extensions to Backus–Naur form are commonly used to define phrase-structure grammars. In one such extension, a question mark (?) indicates that the symbol, or group of symbols inside parentheses, to its left can appear zero or once (that is, it is optional), an asterisk (\*) indicates that the symbol to its left can appear zero or more times, and a plus (+) indicates that the symbol to its left can appear one or more times. These extensions are part of **extended Backus–Naur form (EBNF)**, and the symbols ?, \*, and + are called **metacharacters**. In EBNF the brackets used to denote nonterminals are usually not shown.

34. Describe the set of strings defined by each of these sets of productions in EBNF.

a)  $\text{string} ::= L + D?L+$   
 $L ::= a \mid b \mid c$   
 $D ::= 0 \mid 1$

b)  $\text{string} ::= \text{sign } D+ \mid D+$   
 $\text{sign} ::= + \mid -$   
 $D ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

c)  $\text{string} ::= L*(D+)?L*$   
 $L ::= x \mid y$   
 $D ::= 0 \mid 1$

35. Give production rules in extended Backus–Naur form that generate all decimal numerals consisting of an optional sign, a nonnegative integer, and a decimal fraction that is either the empty string or a decimal point followed by an optional positive integer optionally preceded by some number of zeros.

36. Give production rules in extended Backus–Naur form that generate a sandwich if a sandwich consists of a lower slice of bread; mustard or mayonnaise; optional lettuce; an optional slice of tomato; one or more slices of either turkey, chicken, or roast beef (in any combination); optionally some number of slices of cheese; and a top slice of bread.

37. Give production rules in extended Backus–Naur form for identifiers in the C programming language (see Exercise 33).

38. Describe how productions for a grammar in extended Backus–Naur form can be translated into a set of productions for the grammar in Backus–Naur form.

This is the Backus–Naur form that describes the syntax of expressions in postfix (or reverse Polish) notation.

```
(expression) ::= {term} | {term}{term}{addOperator}
(addOperator) ::= + | -
(term) ::= {factor} | {factor}{factor}{mulOperator}
(mulOperator) ::= * | /
(factor) ::= {identifier} | {expression}
(identifier) ::= a | b | ... | z
```

39. For each of these strings, determine whether it is generated by the grammar given for postfix notation. If it is, find the steps used to generate the string

|          |             |
|----------|-------------|
| a) abc*+ | b) xy++     |
| c) xy-z* | d) wxyz-* / |
| e) ade-* |             |

40. Use Backus–Naur form to describe the syntax of expressions in infix notation, where the set of operators and identifiers is the same as in the BNF for postfix expressions given in the preamble to Exercise 39, but parentheses must surround expressions being used as factors.

41. For each of these strings, determine whether it is generated by the grammar for infix expressions from Exercise 40. If it is, find the steps used to generate the string.

|                        |                      |
|------------------------|----------------------|
| a) $x + y + z$         | b) $a/b + c/d$       |
| c) $m * (n + p)$       | d) $+ m - n + p - q$ |
| e) $(m + n) * (p - q)$ |                      |

42. Let  $G$  be a grammar and let  $R$  be the relation containing the ordered pair  $(w_0, w_1)$  if and only if  $w_1$  is directly derivable from  $w_0$  in  $G$ . What is the reflexive transitive closure of  $R$ ?

## 13.2 Finite-State Machines with Output

### Introduction



Many kinds of machines, including components in computers, can be modeled using a structure called a finite-state machine. Several types of finite-state machines are commonly used in models. All these versions of finite-state machines include a finite set of states, with a designated starting state, an input alphabet, and a transition function that assigns a next state to every state and input pair. Finite-state machines are used extensively in applications in computer science and data networking. For example, finite-state machines are the basis for programs for spell checking, grammar checking, indexing or searching large bodies of text, recognizing speech, transforming text using markup languages such as XML and HTML, and network protocols that specify how computers communicate.

In this section, we will study those finite-state machines that produce output. We will show how finite-state machines can be used to model a vending machine, a machine that delays input, a machine that adds integers, and a machine that determines whether a bit string contains a specified pattern.

**TABLE 1** State Table for a Vending Machine.

| State | Next State |       |       |       |       | Output |    |    |    |    |
|-------|------------|-------|-------|-------|-------|--------|----|----|----|----|
|       | Input      |       |       |       |       | Input  |    |    |    |    |
|       | 5          | 10    | 25    | O     | R     | 5      | 10 | 25 | O  | R  |
| $s_0$ | $s_1$      | $s_2$ | $s_5$ | $s_0$ | $s_0$ | n      | n  | n  | n  | n  |
| $s_1$ | $s_2$      | $s_3$ | $s_6$ | $s_1$ | $s_1$ | n      | n  | n  | n  | n  |
| $s_2$ | $s_3$      | $s_4$ | $s_6$ | $s_2$ | $s_2$ | n      | n  | 5  | n  | n  |
| $s_3$ | $s_4$      | $s_5$ | $s_6$ | $s_3$ | $s_3$ | n      | n  | 10 | n  | n  |
| $s_4$ | $s_5$      | $s_6$ | $s_6$ | $s_4$ | $s_4$ | n      | n  | 15 | n  | n  |
| $s_5$ | $s_6$      | $s_6$ | $s_6$ | $s_5$ | $s_5$ | n      | 5  | 20 | n  | n  |
| $s_6$ | $s_6$      | $s_6$ | $s_6$ | $s_0$ | $s_0$ | 5      | 10 | 25 | OJ | AJ |

Before giving formal definitions, we will show how a vending machine can be modeled. A vending machine accepts nickels (5 cents), dimes (10 cents), and quarters (25 cents). When a total of 30 cents or more has been deposited, the machine immediately returns the amount in excess of 30 cents. When 30 cents has been deposited and any excess refunded, the customer can push an orange button and receive an orange juice or push a red button and receive an apple juice. We can describe how the machine works by specifying its states, how it changes states when input is received, and the output that is produced for every combination of input and current state.

The machine can be in any of seven different states  $s_i$ ,  $i = 0, 1, 2, \dots, 6$ , where  $s_i$  is the state where the machine has collected  $5i$  cents. The machine starts in state  $s_0$ , with 0 cents received. The possible inputs are 5 cents, 10 cents, 25 cents, the orange button ( $O$ ), and the red button ( $R$ ). The possible outputs are nothing ( $n$ ), 5 cents, 10 cents, 15 cents, 20 cents, 25 cents, an orange juice, and an apple juice.

We illustrate how this model of the machine works with this example. Suppose that a student puts in a dime followed by a quarter, receives 5 cents back, and then pushes the orange button for an orange juice. The machine starts in state  $s_0$ . The first input is 10 cents, which changes the state of the machine to  $s_2$  and gives no output. The second input is 25 cents. This changes the state from  $s_2$  to  $s_6$ , and gives 5 cents as output. The next input is the orange button, which changes the state from  $s_6$  back to  $s_0$  (because the machine returns to the start state) and gives an orange juice as its output.

We can display all the state changes and output of this machine in a table. To do this we need to specify for each combination of state and input the next state and the output obtained. Table 1 shows the transitions and outputs for each pair of a state and an input.

Another way to show the actions of a machine is to use a directed graph with labeled edges, where each state is represented by a circle, edges represent the transitions, and edges are labeled with the input and the output for that transition. Figure 1 shows such a directed graph for the vending machine.

## Finite-State Machines with Outputs

We will now give the formal definition of a finite-state machine with output.

### DEFINITION 1

A *finite-state machine*  $M = (S, I, O, f, g, s_0)$  consists of a finite set  $S$  of *states*, a finite *input alphabet*  $I$ , a finite *output alphabet*  $O$ , a *transition function*  $f$  that assigns to each state and input pair a new state, an *output function*  $g$  that assigns to each state and input pair an output, and an *initial state*  $s_0$ .

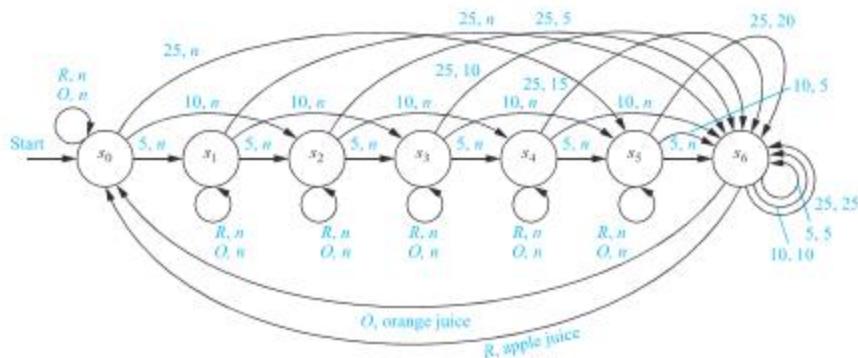


FIGURE 1 A Vending Machine.

Let  $M = (S, I, O, f, g, s_0)$  be a finite-state machine. We can use a **state table** to represent the values of the transition function  $f$  and the output function  $g$  for all pairs of states and input. We previously constructed a state table for the vending machine discussed in the introduction to this section.

**EXAMPLE 1** The state table shown in Table 2 describes a finite-state machine with  $S = \{s_0, s_1, s_2, s_3\}$ ,  $I = \{0, 1\}$ , and  $O = \{0, 1\}$ . The values of the transition function  $f$  are displayed in the first two columns, and the values of the output function  $g$  are displayed in the last two columns.  $\blacktriangleleft$

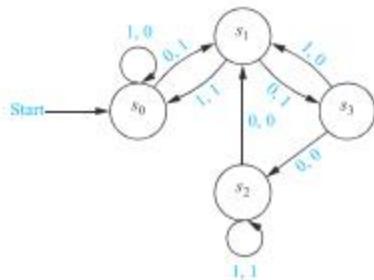
Another way to represent a finite-state machine is to use a **state diagram**, which is a directed graph with labeled edges. In this diagram, each state is represented by a circle. Arrows labeled with the input and output pair are shown for each transition.

**EXAMPLE 2** Construct the state diagram for the finite-state machine with the state table shown in Table 2.  $\blacktriangleleft$

*Solution:* The state diagram for this machine is shown in Figure 2.  $\blacktriangleleft$

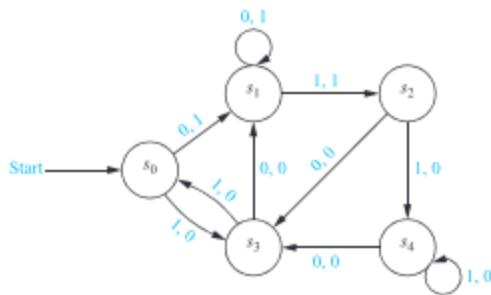
**EXAMPLE 3** Construct the state table for the finite-state machine with the state diagram shown in Figure 3.  $\blacktriangleleft$

*Solution:* The state table for this machine is shown in Table 3.  $\blacktriangleleft$



| State | $f$   |       | $g$   |   |
|-------|-------|-------|-------|---|
|       | Input |       | Input |   |
|       | 0     | 1     | 0     | 1 |
| $s_0$ | $s_1$ | $s_0$ | 1     | 0 |
| $s_1$ | $s_3$ | $s_0$ | 1     | 1 |
| $s_2$ | $s_1$ | $s_2$ | 0     | 1 |
| $s_3$ | $s_2$ | $s_1$ | 0     | 0 |

FIGURE 2 The State Diagram for the Finite-State Machine Shown in Table 2.

**FIGURE 3** A Finite-State Machine.

An input string takes the starting state through a sequence of states, as determined by the transition function. As we read the input string symbol by symbol (from left to right), each input symbol takes the machine from one state to another. Because each transition produces an output, an input string also produces an output string.

Suppose that the input string is  $x = x_1x_2 \dots x_k$ . Then, reading this input takes the machine from state  $s_0$  to state  $s_1$ , where  $s_1 = f(s_0, x_1)$ , then to state  $s_2$ , where  $s_2 = f(s_1, x_2)$ , and so on, with  $s_j = f(s_{j-1}, x_j)$  for  $j = 1, 2, \dots, k$ , ending at state  $s_k = f(s_{k-1}, x_k)$ . This sequence of transitions produces an output string  $y_1y_2 \dots y_k$ , where  $y_1 = g(s_0, x_1)$  is the output corresponding to the transition from  $s_0$  to  $s_1$ ,  $y_2 = g(s_1, x_2)$  is the output corresponding to the transition from  $s_1$  to  $s_2$ , and so on. In general,  $y_j = g(s_{j-1}, x_j)$  for  $j = 1, 2, \dots, k$ . Hence, we can extend the definition of the output function  $g$  to input strings so that  $g(x) = y$ , where  $y$  is the output corresponding to the input string  $x$ . This notation is useful in many applications.

**EXAMPLE 4** Find the output string generated by the finite-state machine in Figure 3 if the input string is 101011.

**Solution:** The output obtained is 001000. The successive states and outputs are shown in Table 4. ◀

We can now look at some examples of useful finite-state machines. Examples 5, 6, and 7 illustrate that the states of a finite-state machine give it limited memory capabilities. The states can be used to remember the properties of the symbols that have been read by the machine. However, because there are only finitely many different states, finite-state machines cannot be used for some important purposes. This will be illustrated in Section 13.4.

**EXAMPLE 5** An important element in many electronic devices is a *unit-delay machine*, which produces as output the input string delayed by a specified amount of time. How can a finite-state machine be constructed that delays an input string by one unit of time, that is, produces as output the bit string  $0x_1x_2 \dots x_{k-1}$  given the input bit string  $x_1x_2 \dots x_k$ ?

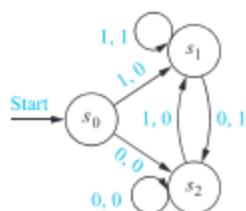
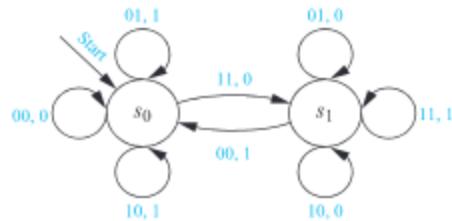
**Solution:** A delay machine can be constructed that has two possible inputs, namely, 0 and 1. The machine must have a start state  $s_0$ . Because the machine has to remember whether the previous

**TABLE 3**

| State | f          |            | g          |            |
|-------|------------|------------|------------|------------|
|       | Input<br>0 | Input<br>1 | Input<br>0 | Input<br>1 |
| $s_0$ | $s_1$      | $s_3$      | 1          | 0          |
| $s_1$ | $s_1$      | $s_2$      | 1          | 1          |
| $s_2$ | $s_3$      | $s_4$      | 0          | 0          |
| $s_3$ | $s_1$      | $s_0$      | 0          | 0          |
| $s_4$ | $s_3$      | $s_4$      | 0          | 0          |

**TABLE 4**

| Input  | 1     | 0     | 1     | 0     | 1     | 1     | —     |
|--------|-------|-------|-------|-------|-------|-------|-------|
| State  | $s_0$ | $s_3$ | $s_1$ | $s_2$ | $s_3$ | $s_0$ | $s_3$ |
| Output | 0     | 0     | 1     | 0     | 0     | 0     | —     |

**FIGURE 4** A Unit-Delay Machine.**FIGURE 5** A Finite-State Machine for Addition.

input was a 0 or a 1, two other states  $s_1$  and  $s_2$  are needed, where the machine is in state  $s_1$  if the previous input was 1 and in state  $s_2$  if the previous input was 0. An output of 0 is produced for the initial transition from  $s_0$ . Each transition from  $s_1$  gives an output of 1, and each transition from  $s_2$  gives an output of 0. The output corresponding to the input of a string  $x_1 \dots x_k$  is the string that begins with 0, followed by  $x_1$ , followed by  $x_2, \dots$ , ending with  $x_{k-1}$ . The state diagram for this machine is shown in Figure 4.  $\blacktriangleleft$

**EXAMPLE 6** Produce a finite-state machine that adds two positive integers using their binary expansions.



**Solution:** When  $(x_n \dots x_1 x_0)_2$  and  $(y_n \dots y_1 y_0)_2$  are added, the following procedure (as described in Section 4.2) is followed. First, the bits  $x_0$  and  $y_0$  are added, producing a sum bit  $z_0$  and a carry bit  $c_0$ . This carry bit is either 0 or 1. Then, the bits  $x_1$  and  $y_1$  are added, together with the carry  $c_0$ . This gives a sum bit  $z_1$  and a carry bit  $c_1$ . This procedure is continued until the  $n$ th stage, where  $x_n$ ,  $y_n$ , and the previous carry  $c_{n-1}$  are added to produce the sum bit  $z_n$  and the carry bit  $c_n$ , which is equal to the sum bit  $z_{n+1}$ .

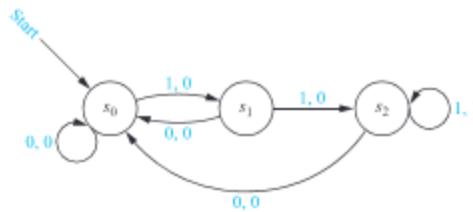
A finite-state machine to carry out this addition can be constructed using just two states. For simplicity we assume that both the initial bits  $x_n$  and  $y_n$  are 0 (otherwise we have to make special arrangements concerning the sum bit  $z_{n+1}$ ). The start state  $s_0$  is used to remember that the previous carry is 0 (or for the addition of the rightmost bits). The other state,  $s_1$ , is used to remember that the previous carry is 1.

Because the inputs to the machine are pairs of bits, there are four possible inputs. We represent these possibilities by 00 (when both bits are 0), 01 (when the first bit is 0 and the second is 1), 10 (when the first bit is 1 and the second is 0), and 11 (when both bits are 1). The transitions and the outputs are constructed from the sum of the two bits represented by the input and the carry represented by the state. For instance, when the machine is in state  $s_1$  and receives 01 as input, the next state is  $s_1$  and the output is 0, because the sum that arises is  $0 + 1 + 1 = (10)_2$ . The state diagram for this machine is shown in Figure 5.  $\blacktriangleleft$

**EXAMPLE 7** In a certain coding scheme, when three consecutive 1s appear in a message, the receiver of the message knows that there has been a transmission error. Construct a finite-state machine that gives a 1 as its current output bit if and only if the last three bits received are all 1s.

**Solution:** Three states are needed in this machine. The start state  $s_0$  remembers that the previous input value, if it exists, was not a 1. The state  $s_1$  remembers that the previous input was a 1, but the input before the previous input, if it exists, was not a 1. The state  $s_2$  remembers that the previous two inputs were 1s.

An input of 1 takes  $s_0$  to  $s_1$ , because now a 1, and not two consecutive 1s, has been read; it takes  $s_1$  to  $s_2$ , because now two consecutive 1s have been read; and it takes  $s_2$  to itself, because at least two consecutive 1s have been read. An input of 0 takes every state to  $s_0$ , because this breaks up any string of consecutive 1s. The output for the transition from  $s_2$  to itself when a 1



**FIGURE 6** A Finite-State Machine That Gives an Output of 1 If and Only If the Input String Read So Far Ends with 111.

is read is 1, because this combination of input and state shows that three consecutive 1s have been read. All other outputs are 0. The state diagram of this machine is shown in Figure 6.  $\blacktriangleleft$

The final output bit of the finite-state machine we constructed in Example 7 is 1 if and only if the input string ends with 111. Because of this, we say that this finite-state machine **recognizes** the set of bit strings that end with 111. This leads us to Definition 2.

### DEFINITION 2

Let  $M = (S, I, O, f, g, s_0)$  be a finite-state machine and  $L \subseteq I^*$ . We say that  $M$  *recognizes* (or *accepts*)  $L$  if an input string  $x$  belongs to  $L$  if and only if the last output bit produced by  $M$  when given  $x$  as input is a 1.

**TYPES OF FINITE-STATE MACHINES** Many different kinds of finite-state machines have been developed to model computing machines. In this section we have given a definition of one type of finite-state machine. In the type of machine introduced in this section, outputs correspond to transitions between states. Machines of this type are known as **Mealy machines**, because they were first studied by G. H. Mealy in 1955. There is another important type of finite-state machine with output, where the output is determined only by the state. This type of finite-state machine is known as a **Moore machine**, because E. F. Moore introduced this type of machine in 1956. Moore machines are considered in a sequence of exercises.

In Example 7 we showed how a Mealy machine can be used for language recognition. However, another type of finite-state machine, giving no output, is usually used for this purpose. Finite-state machines with no output, also known as finite-state automata, have a set of final states and recognize a string if and only if it takes the start state to a final state. We will study this type of finite-state machine in Section 13.3.

## Exercises

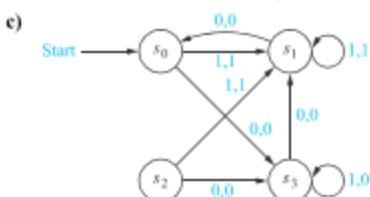
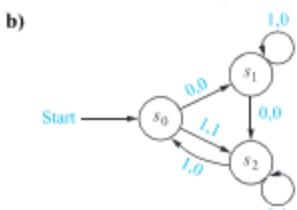
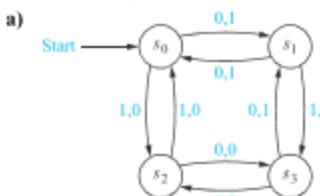
1. Draw the state diagrams for the finite-state machines with these state tables.

| State | $f$   |       | $g$   |   |
|-------|-------|-------|-------|---|
|       | Input |       | Input |   |
|       | 0     | 1     | 0     | 1 |
| $s_0$ | $s_1$ | $s_0$ | 0     | 1 |
| $s_1$ | $s_0$ | $s_2$ | 0     | 1 |
| $s_2$ | $s_1$ | $s_1$ | 0     | 0 |

| State | $f$   |       | $g$   |   |
|-------|-------|-------|-------|---|
|       | Input |       | Input |   |
|       | 0     | 1     | 0     | 1 |
| $s_0$ | $s_1$ | $s_0$ | 0     | 0 |
| $s_1$ | $s_2$ | $s_0$ | 1     | 1 |
| $s_2$ | $s_0$ | $s_3$ | 0     | 1 |
| $s_3$ | $s_1$ | $s_2$ | 1     | 0 |

| State | <i>f</i>     |       | <i>g</i>     |   |
|-------|--------------|-------|--------------|---|
|       | <i>Input</i> |       | <i>Input</i> |   |
|       | 0            | 1     | 0            | 1 |
| $s_0$ | $s_0$        | $s_4$ | 1            | 1 |
| $s_1$ | $s_0$        | $s_3$ | 0            | 1 |
| $s_2$ | $s_0$        | $s_2$ | 0            | 0 |
| $s_3$ | $s_1$        | $s_1$ | 1            | 1 |
| $s_4$ | $s_1$        | $s_0$ | 1            | 0 |

2. Give the state tables for the finite-state machines with these state diagrams.



3. Find the output generated from the input string 01110 for the finite-state machine with the state table in

- a) Exercise 1(a).
- b) Exercise 1(b).
- c) Exercise 1(c).

4. Find the output generated from the input string 10001 for the finite-state machine with the state diagram in

- a) Exercise 2(a).
- b) Exercise 2(b).
- c) Exercise 2(c).

5. Find the output for each of these input strings when given as input to the finite-state machine in Example 2.

a) 0111      b) 11011011      c) 01010101010

6. Find the output for each of these input strings when given as input to the finite-state machine in Example 3.

a) 0000      b) 101010      c) 11011100010

7. Construct a finite-state machine that models an old-fashioned soda machine that accepts nickels, dimes, and quarters. The soda machine accepts change until 35 cents has been put in. It gives change back for any amount greater than 35 cents. Then the customer can push buttons to receive either a cola, a root beer, or a ginger ale.

8. Construct a finite-state machine that models a newspaper vending machine that has a door that can be opened only after either three dimes (and any number of other coins) or a quarter and a nickel (and any number of other coins) have been inserted. Once the door can be opened, the customer opens it and takes a paper, closing the door. No change is ever returned no matter how much extra money has been inserted. The next customer starts with no credit.

9. Construct a finite-state machine that delays an input string two bits, giving 00 as the first two bits of output.

10. Construct a finite-state machine that changes every other bit, starting with the second bit, of an input string, and leaves the other bits unchanged.

11. Construct a finite-state machine for the log-on procedure for a computer, where the user logs on by entering a user identification number, which is considered to be a single input, and then a password, which is considered to be a single input. If the password is incorrect, the user is asked for the user identification number again.

12. Construct a finite-state machine for a combination lock that contains numbers 1 through 40 and that opens only when the correct combination, 10 right, 8 second left, 37 right, is entered. Each input is a triple consisting of a number, the direction of the turn, and the number of times the lock is turned in that direction.

13. Construct a finite-state machine for a toll machine that opens a gate after 25 cents, in nickels, dimes, or quarters, has been deposited. No change is given for overpayment, and no credit is given to the next driver when more than 25 cents has been deposited.

14. Construct a finite-state machine for entering a security code into an automatic teller machine (ATM) that implements these rules: A user enters a string of four digits, one digit at a time. If the user enters the correct four digits of the password, the ATM displays a welcome screen. When the user enters an incorrect string of four digits, the ATM displays a screen that informs the user that an incorrect password was entered. If a user enters the incorrect password three times, the account is locked.

15. Construct a finite-state machine for a restricted telephone switching system that implements these rules. Only calls to the telephone numbers 0, 911, and the digit 1 followed by 10-digit telephone numbers that begin with 212, 800, 866, 877, and 888 are sent to the network. All other strings of digits are blocked by the system and the user hears an error message.

16. Construct a finite-state machine that gives an output of 1 if the number of input symbols read so far is divisible by 3 and an output of 0 otherwise.

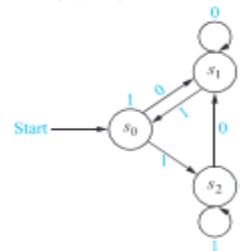
17. Construct a finite-state machine that determines whether the input string has a 1 in the last position and a 0 in the third to the last position read so far.
18. Construct a finite-state machine that determines whether the input string read so far ends in at least five consecutive 1s.
19. Construct a finite-state machine that determines whether the word *computer* has been read as the last eight characters in the input read so far, where the input can be any string of English letters.

A **Moore machine**  $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, f, g, s_0)$  consists of a finite set of states, an input alphabet  $\mathcal{I}$ , an output alphabet  $\mathcal{O}$ , a transition function  $f$  that assigns a next state to every pair of a state and an input, an output function  $g$  that assigns an output to every state, and a starting state  $s_0$ . A Moore machine can be represented either by a table listing the transitions for each pair of state and input and the outputs for each state, or by a state diagram that displays the states, the transitions between states, and the output for each state. In the diagram, transitions are indicated with arrows labeled with the input, and the outputs are shown next to the states.

20. Construct the state diagram for the Moore machine with this state table.

| State | $f$     |         | g |
|-------|---------|---------|---|
|       | Input 0 | Input 1 |   |
| $s_0$ | $s_0$   | $s_2$   | 0 |
| $s_1$ | $s_3$   | $s_0$   | 1 |
| $s_2$ | $s_2$   | $s_1$   | 1 |
| $s_3$ | $s_2$   | $s_0$   | 1 |

21. Construct the state table for the Moore machine with the state diagram shown here. Each input string to a Moore machine  $M$  produces an output string. In particular, the output corresponding to the input string  $a_1a_2\dots a_k$  is the string  $g(s_0)g(s_1)\dots g(s_k)$ , where  $s_i = f(s_{i-1}, a_i)$  for  $i = 1, 2, \dots, k$ .



22. Find the output string generated by the Moore machine in Exercise 20 with each of these input strings.
  - a) 0101
  - b) 111111
  - c) 11101110111
23. Find the output string generated by the Moore machine in Exercise 21 with each of the input strings in Exercise 22.
24. Construct a Moore machine that gives an output of 1 whenever the number of symbols in the input string read so far is divisible by 4 and an output of 0 otherwise.
25. Construct a Moore machine that determines whether an input string contains an even or odd number of 1s. The machine should give 1 as output if an even number of 1s are in the string and 0 as output if an odd number of 1s are in the string.

## 13.3 Finite-State Machines with No Output

### Introduction



One of the most important applications of finite-state machines is in language recognition. This application plays a fundamental role in the design and construction of compilers for programming languages. In Section 13.2 we showed that a finite-state machine with output can be used to recognize a language, by giving an output of 1 when a string from the language has been read and a 0 otherwise. However, there are other types of finite-state machines that are specially designed for recognizing languages. Instead of producing output, these machines have final states. A string is recognized if and only if it takes the starting state to one of these final states.

### Set of Strings

Before discussing finite-state machines with no output, we will introduce some important background material on sets of strings. The operations that will be defined here will be used extensively in our discussion of language recognition by finite-state machines.

**DEFINITION 1**

Suppose that  $A$  and  $B$  are subsets of  $V^*$ , where  $V$  is a vocabulary. The *concatenation* of  $A$  and  $B$ , denoted by  $AB$ , is the set of all strings of the form  $xy$ , where  $x$  is a string in  $A$  and  $y$  is a string in  $B$ .

**EXAMPLE 1** Let  $A = \{0, 11\}$  and  $B = \{1, 10, 110\}$ . Find  $AB$  and  $BA$ .

*Solution:* The set  $AB$  contains every concatenation of a string in  $A$  and a string in  $B$ . Hence,  $AB = \{01, 010, 0110, 111, 1110, 11110\}$ . The set  $BA$  contains every concatenation of a string in  $B$  and a string in  $A$ . Hence,  $BA = \{10, 111, 100, 1011, 1100, 11011\}$ . 

Note that it is not necessarily the case that  $AB = BA$  when  $A$  and  $B$  are subsets of  $V^*$ , where  $V$  is an alphabet, as Example 1 illustrates.

From the definition of the concatenation of two sets of strings, we can define  $A^n$ , for  $n = 0, 1, 2, \dots$ . This is done recursively by specifying that

$$\begin{aligned} A^0 &= \{\lambda\}, \\ A^{n+1} &= A^n A \quad \text{for } n = 0, 1, 2, \dots \end{aligned}$$

**EXAMPLE 2** Let  $A = \{1, 00\}$ . Find  $A^n$  for  $n = 0, 1, 2$ , and  $3$ .

*Solution:* We have  $A^0 = \{\lambda\}$  and  $A^1 = A^0 A = \{\lambda\} A = \{1, 00\}$ . To find  $A^2$  we take concatenations of pairs of elements of  $A$ . This gives  $A^2 = \{11, 100, 001, 0000\}$ . To find  $A^3$  we take concatenations of elements in  $A^2$  and  $A$ ; this gives  $A^3 = \{111, 1100, 1001, 10000, 0011, 00100, 00001, 000000\}$ . 

**DEFINITION 2**

Suppose that  $A$  is a subset of  $V^*$ . Then the *Kleene closure* of  $A$ , denoted by  $A^*$ , is the set consisting of concatenations of arbitrarily many strings from  $A$ . That is,  $A^* = \bigcup_{k=0}^{\infty} A^k$ .

**EXAMPLE 3** What are the Kleene closures of the sets  $A = \{0\}$ ,  $B = \{0, 1\}$ , and  $C = \{11\}$ ?

*Solution:* The Kleene closure of  $A$  is the concatenation of the string  $0$  with itself an arbitrary finite number of times. Hence,  $A^* = \{0^n \mid n = 0, 1, 2, \dots\}$ . The Kleene closure of  $B$  is the concatenation of an arbitrary number of strings, where each string is either  $0$  or  $1$ . This is the set of all strings over the alphabet  $V = \{0, 1\}$ . That is,  $B^* = V^*$ . Finally, the Kleene closure of  $C$  is the concatenation of the string  $11$  with itself an arbitrary number of times. Hence,  $C^*$  is the set of strings consisting of an even number of  $1$ s. That is,  $C^* = \{1^{2n} \mid n = 0, 1, 2, \dots\}$ . 

## Finite-State Automata

We will now give a definition of a finite-state machine with no output. Such machines are also called **finite-state automata**, and that is the terminology we will use for them here. (Note: The singular of *automata* is *automaton*.) These machines differ from the finite-state machines studied in Section 13.2 in that they do not produce output, but they do have a set of final states. As we will see, they recognize strings that take the starting state to a final state.

| State | <i>f</i> |       |
|-------|----------|-------|
|       | Input    |       |
|       | 0        | 1     |
| $s_0$ | $s_0$    | $s_1$ |
| $s_1$ | $s_0$    | $s_2$ |
| $s_2$ | $s_0$    | $s_0$ |
| $s_3$ | $s_2$    | $s_1$ |

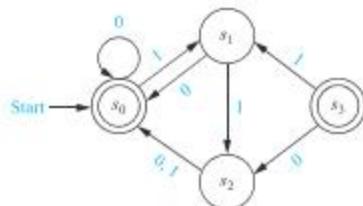


FIGURE 1 The State Diagram for a Finite-State Automaton.

### DEFINITION 3

A *finite-state automaton*  $M = (S, I, f, s_0, F)$  consists of a finite set  $S$  of states, a finite input alphabet  $I$ , a transition function  $f$  that assigns a next state to every pair of state and input (so that  $f : S \times I \rightarrow S$ ), an initial or start state  $s_0$ , and a subset  $F$  of  $S$  consisting of final (or accepting) states.

We can represent finite-state automata using either state tables or state diagrams. Final states are indicated in state diagrams by using double circles.

**EXAMPLE 4** Construct the state diagram for the finite-state automaton  $M = (S, I, f, s_0, F)$ , where  $S = \{s_0, s_1, s_2, s_3\}$ ,  $I = \{0, 1\}$ ,  $F = \{s_0, s_3\}$ , and the transition function  $f$  is given in Table 1.

**Solution:** The state diagram is shown in Figure 1. Note that because both the inputs 0 and 1 take  $s_2$  to  $s_0$ , we write 0,1 over the edge from  $s_2$  to  $s_0$ . ▲

**EXTENDING THE TRANSITION FUNCTION** The transition function  $f$  of a finite-state machine  $M = (S, I, f, s_0, F)$  can be extended so that it is defined for all pairs of states and strings; that is,  $f$  can be extended to a function  $f : S \times I^* \rightarrow S$ . Let  $x = x_1x_2\dots x_k$  be a string in  $I^*$ . Then  $f(s_1, x)$  is the state obtained by using each successive symbol of  $x$ , from left to right, as input, starting with state  $s_1$ . From  $s_1$  we go on to state  $s_2 = f(s_1, x_1)$ , then to state  $s_3 = f(s_2, x_2)$ , and so on, with  $f(s_1, x) = f(s_k, x_1)$ . Formally, we can define this extended transition function  $f$  recursively for the deterministic finite-state machine  $M = (S, I, f, s_0, F)$  by

- (i)  $f(s, \lambda) = s$  for every state  $s \in S$ ; and
- (ii)  $f(s, xa) = f(f(s, x), a)$  for all  $s \in S, x \in I^*$ , and  $a \in I$ .



**STEPHEN COLE KLEENE (1909–1994)** Stephen Kleene was born in Hartford, Connecticut. His mother, Alice Lena Cole, was a poet, and his father, Gustav Adolph Kleene, was an economics professor. Kleene attended Amherst College and received his Ph.D. from Princeton in 1934, where he studied under the famous logician Alonzo Church. Kleene joined the faculty of the University of Wisconsin in 1935, where he remained except for several leaves, including stays at the Institute for Advanced Study in Princeton. During World War II he was a navigation instructor at the Naval Reserve's Midshipmen's School and later served as the director of the Naval Research Laboratory. Kleene made significant contributions to the theory of recursive functions, investigating questions of computability and decidability, and proved one of the central results of automata theory. He served as the Acting Director of the Mathematics Research Center and as Dean of the College of Letters and Sciences at the University of Wisconsin. Kleene was a student of natural history. He discovered a previously undescribed variety of butterfly that is named after him. He was an avid hiker and climber. Kleene was also noted as a talented teller of anecdotes, using a powerful voice that could be heard several offices away.

We can use structural induction and this recursive definition to prove properties of this extended transition function. For example, in Exercise 15 we ask you to prove that

$$f(s, xy) = f(f(s, x), y)$$

for every state  $s \in S$  and strings  $x \in I^*$  and  $y \in I^*$ .

## Language Recognition by Finite-State Machines

Next, we define some terms that are used when studying the recognition by finite-state automata of certain sets of strings.

### DEFINITION 4

A string  $x$  is said to be *recognized* or *accepted* by the machine  $M = (S, I, f, s_0, F)$  if it takes the initial state  $s_0$  to a final state, that is,  $f(s_0, x)$  is a state in  $F$ . The *language recognized* or *accepted* by the machine  $M$ , denoted by  $L(M)$ , is the set of all strings that are recognized by  $M$ . Two finite-state automata are called *equivalent* if they recognize the same language.

In Example 5 we will find the languages recognized by several finite-state automata.

**EXAMPLE 5** Determine the languages recognized by the finite-state automata  $M_1$ ,  $M_2$ , and  $M_3$  in Figure 2.

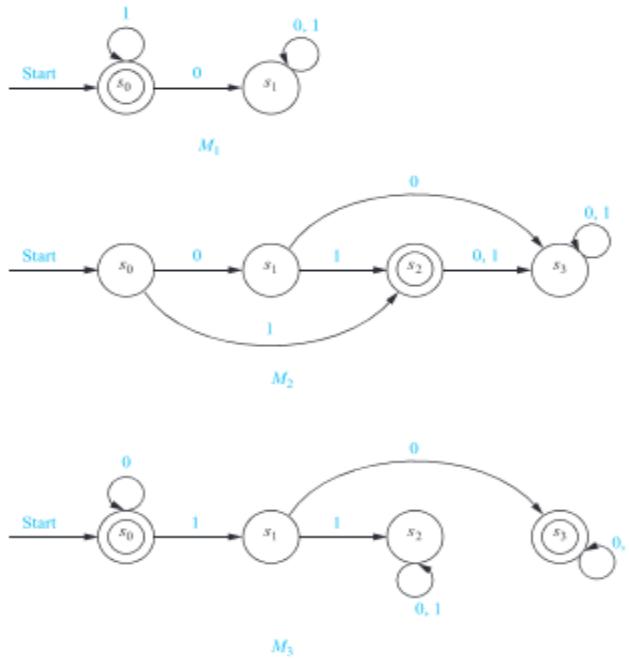


FIGURE 2 Some Finite-State Automata.

**Solution:** The only final state of  $M_1$  is  $s_0$ . The strings that take  $s_0$  to itself are those consisting of zero or more consecutive 1s. Hence,  $L(M_1) = \{1^n \mid n = 0, 1, 2, \dots\}$ .

The only final state of  $M_2$  is  $s_2$ . The only strings that take  $s_0$  to  $s_2$  are 1 and 01. Hence,  $L(M_2) = \{1, 01\}$ .

The final states of  $M_3$  are  $s_0$  and  $s_3$ . The only strings that take  $s_0$  to itself are  $\lambda, 0, 00, 000, \dots$ , that is, any string of zero or more consecutive 0s. The only strings that take  $s_0$  to  $s_3$  are a string of zero or more consecutive 0s, followed by 10, followed by any string. Hence,  $L(M_3) = \{0^n, 0^n 10x \mid n = 0, 1, 2, \dots, \text{and } x \text{ is any string}\}$ .  $\blacktriangleleft$

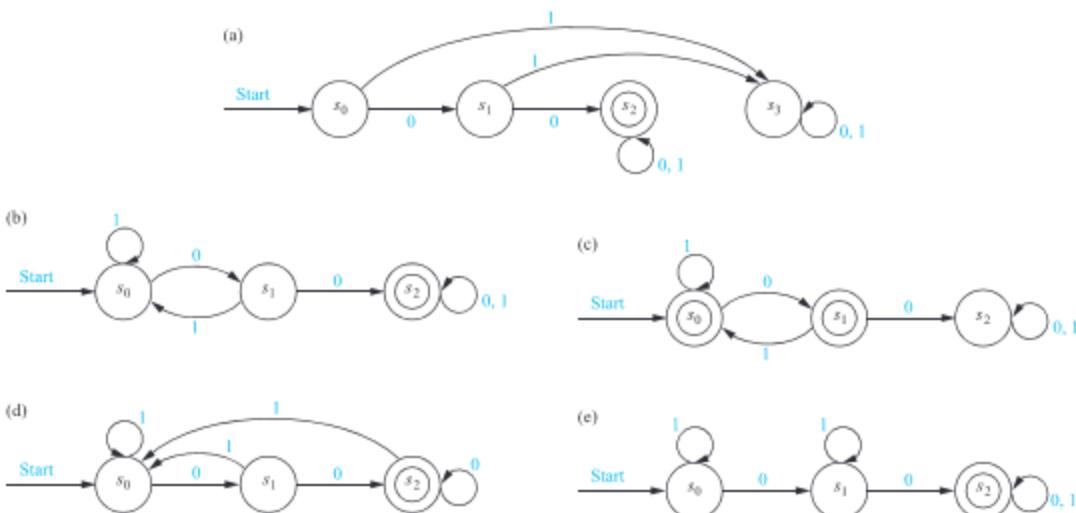
**DESIGNING FINITE-STATE AUTOMATA** We can often construct a finite-state automaton that recognizes a given set of strings by carefully adding states and transitions and determining which of these states should be final states. When appropriate we include states that can keep track of some of the properties of the input string, providing the finite-state automaton with limited memory. Examples 6 and 7 illustrate some of the techniques that can be used to construct finite-state automata that recognize particular types of sets of strings.

**EXAMPLE 6** Construct deterministic finite-state automata that recognize each of these languages.



- the set of bit strings that begin with two 0s
- the set of bit strings that contain two consecutive 0s
- the set of bit strings that do not contain two consecutive 0s
- the set of bit strings that end with two 0s
- the set of bit strings that contain at least two 0s

**Solution:** (a) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that begin with two 0s. Besides the start state  $s_0$ , we include a nonfinal state  $s_1$ ; we move to  $s_1$  from  $s_0$  if the first bit is a 0. Next, we add a final state  $s_2$ , which we move to from  $s_1$  if the second bit is a 0. When we have reached  $s_2$  we know that the first two input bits are both 0s, so we stay in the state  $s_2$  no matter what the succeeding bits (if any) are. We move to a nonfinal state  $s_3$  from  $s_0$  if the first bit is a 1 and from  $s_1$  if the second bit is a 1. The reader should verify that the finite-state automaton in Figure 3(a) recognizes the set of bit strings that begin with two 0s.



**FIGURE 3** Deterministic Finite-State Automata Recognizing the Languages in Example 6.

(b) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that contain two consecutive 0s. Besides the start state  $s_0$ , we include a nonfinal state  $s_1$ , which tells us that the last input bit seen is a 0, but either the bit before it was a 1, or this bit was the initial bit of the string. We include a final state  $s_2$  that we move to from  $s_1$  when the next input bit after a 0 is also a 0. If a 1 follows a 0 in the string (before we encounter two consecutive 0s), we return to  $s_0$  and begin looking for consecutive 0s all over again. The reader should verify that the finite-state automaton in Figure 3(b) recognizes the set of bit strings that contain two consecutive 0s.

(c) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that do not contain two consecutive 0s. Besides the start state  $s_0$ , which should be a final state, we include a final state  $s_1$ , which we move to from  $s_0$  when 0 is the first input bit. When an input bit is a 1, we return to, or stay in, state  $s_0$ . We add a state  $s_2$ , which we move to from  $s_1$  when the input bit is a 0. Reaching  $s_2$  tells us that we have seen two consecutive 0s as input bits. We stay in state  $s_2$  once we have reached it; this state is not final. The reader should verify that the finite-state automaton in Figure 3(c) recognizes the set of bit strings that do not contain two consecutive 0s. [The astute reader will notice the relationship between the finite-state automaton constructed here and the one constructed in part (b). See Exercise 39.]

(d) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that end with two 0s. Besides the start state  $s_0$ , we include a nonfinal state  $s_1$ , which we move to if the first bit is 0. We include a final state  $s_2$ , which we move to from  $s_1$  if the next input bit after a 0 is also a 0. If an input of 0 follows a previous 0, we stay in state  $s_2$  because the last two input bits are still 0s. Once we are in state  $s_2$ , an input bit of 1 sends us back to  $s_0$ , and we begin looking for consecutive 0s all over again. We also return to  $s_0$  if the next input is a 1 when we are in state  $s_1$ . The reader should verify that the finite-state automaton in Figure 3(d) recognizes the set of bit strings that end with two 0s.

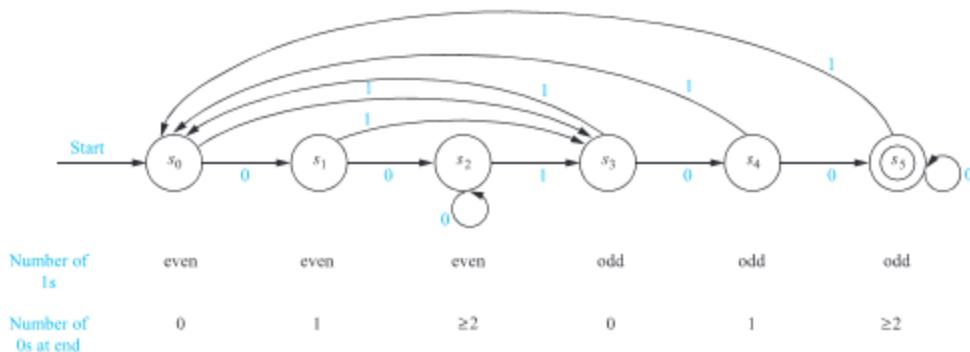
(e) Our goal is to construct a deterministic finite-state automaton that recognizes the set of bit strings that contain two 0s. Besides the start state, we include a state  $s_1$ , which is not final; we stay in  $s_0$  until an input bit is a 0 and we move to  $s_1$  when we encounter the first 0 bit in the input. We add a final state  $s_2$ , which we move to from  $s_1$  once we encounter a second 0 bit. Whenever we encounter a 1 as input, we stay in the current state. Once we have reached  $s_2$ , we remain there. Here,  $s_1$  and  $s_2$  are used to tell us that we have already seen one or two 0s in the input string so far, respectively. The reader should verify that the finite-state automaton in Figure 3(e) recognizes the set of bit strings that contain two 0s. 

**EXAMPLE 7** Construct a deterministic finite-state automaton that recognizes the set of bit strings that contain an odd number of 1s and that end with at least two consecutive 0s.

**Solution:** We can build a deterministic finite-state automaton that recognizes the specified set by including states that keep track of both the parity of the number of 1 bits and whether we have seen no, one, or at least two 0s at the end of the input string.

The start state  $s_0$  can be used to tell us that the input read so far contains an even number of 1s and ends with no 0s (that is, is empty or ends with a 1). Besides the start state, we include five more states. We move to states  $s_1, s_2, s_3, s_4$ , and  $s_5$ , respectively, when the input string read so far contains an even number of 1s and ends with one 0; when it contains an even number of 1s and ends with at least two 0s; when it contains an odd number of 1s and ends with no 0s; when it contains an odd number of 1s and ends with one 0; and when it contains an odd number of 1s and ends with two 0s. The state  $s_5$  is a final state.

The reader should verify that the finite-state automaton in Figure 4 recognizes the set of bit strings that contain an odd number of 1s and end with at least two consecutive 0s. 



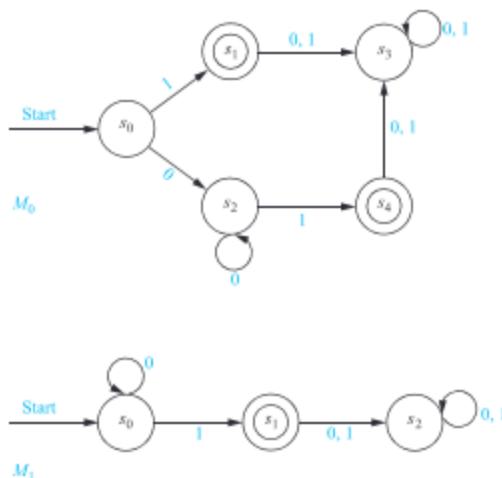
**FIGURE 4** A Deterministic Finite-State Automaton Recognizing the Set of Bit Strings Containing an Odd Number of 1s and Ending with at Least Two 0s.

**EQUIVALENT FINITE-STATE AUTOMATA** In Definition 4 we specified that two finite-state automata are equivalent if they recognize the same language. Example 8 provides an example of two equivalent deterministic finite-state machines.

**EXAMPLE 8** Show that the two finite-state automata  $M_0$  and  $M_1$  shown in Figure 5 are equivalent.

**Solution:** For a string  $x$  to be recognized by  $M_0$ ,  $x$  must take us from  $s_0$  to the final state  $s_1$  or the final state  $s_4$ . The only string that takes us from  $s_0$  to  $s_1$  is the string 1. The strings that take us from  $s_0$  to  $s_4$  are those strings that begin with a 0, which takes us from  $s_0$  to  $s_2$ , followed by zero or more additional 0s, which keep the machine in state  $s_2$ , followed by a 1, which takes us from state  $s_2$  to the final state  $s_4$ . All other strings take us from  $s_0$  to a state that is not final. (We leave it to the reader to fill in the details.) We conclude that  $L(M_0)$  is the set of strings of zero or more 0 bits followed by a final 1.

For a string  $x$  to be recognized by  $M_1$ ,  $x$  must take us from  $s_0$  to the final state  $s_1$ . So, for  $x$  to be recognized, it must begin with some number of 0s, which leave us in state  $s_0$ , followed by



**FIGURE 5**  $M_0$  and  $M_1$  Are Equivalent Finite-State Automata.

a 1, which takes us to the final state  $s_1$ . A string of all zeros is not recognized because it leaves us in state  $s_0$ , which is not final. All strings that contain a 0 after 1 are not recognized because they take us to state  $s_2$ , which is not final. It follows that  $L(M_1)$  is the same as  $L(M_0)$ . We conclude that  $M_0$  and  $M_1$  are equivalent.

Note that the finite-state machine  $M_1$  only has three states. No finite state machine with fewer than three states can be used to recognize the set of all strings of zero or more 0 bits followed by a 1 (see Exercise 37). 

As Example 8 shows, a finite-state automaton may have more states than one equivalent to it. In fact, algorithms used to construct finite-state automata to recognize certain languages may have many more states than necessary. Using unnecessarily large finite-state machines to recognize languages can make both hardware and software applications inefficient and costly. This problem arises when finite-state automata are used in compilers, which translate computer programs to a language a computer can understand (object code).

Exercises 58–61 develop a procedure that constructs a finite-state automaton with the fewest states possible among all finite-state automata equivalent to a given finite-state automaton. This procedure is known as **machine minimization**. The minimization procedure described in these exercises reduces the number of states by replacing states with equivalence classes of states with respect to an equivalence relation in which two states are equivalent if every input string either sends both states to a final state or sends both to a state that is not final. Before the minimization procedure begins, all states that cannot be reached from the start state using any input string are first removed; removing these does not change the language recognized.



**GRACE BREWSTER MURRAY HOPPER (1906–1992)** Grace Hopper, born in New York City, displayed an intense curiosity as a child with how things worked. At the age of seven, she disassembled alarm clocks to discover their mechanisms. She inherited her love of mathematics from her mother, who received special permission to study geometry (but not algebra and trigonometry) at a time when women were actively discouraged from such study. Hopper was inspired by her father, a successful insurance broker, who had lost his legs from circulatory problems. He told his children they could do anything if they put their minds to it. He inspired Hopper to pursue higher education and not conform to the usual roles for females. Her parents made sure that she had an excellent education; she attended private schools for girls in New York. Hopper entered Vassar College in 1924, where she majored in mathematics and physics; she graduated in 1928. She received a masters degree in mathematics from Yale University in 1930. In 1930 she also married an English instructor at the New York School of Commerce; she later divorced and did not have children. Hopper was a mathematics professor at Vassar from 1931 until 1943, earning a Ph.D. from Yale in 1934.

After the attack on Pearl Harbor, Hopper, coming from a family with strong military traditions, decided to leave her academic position and join the Navy WAVES. To enlist, she needed special permission to leave her strategic position as a mathematics professor, as well as a waiver for weighing too little. In December 1943, she was sworn into the Navy Reserve and trained at the Midshipman's School for Women. Hopper was assigned to work at the Naval Ordnance Laboratory] at Harvard University. She wrote programs for the world's first large-scale automatically sequenced digital computer, which was used to help aim Navy artillery in varying weather. Hopper has been credited with coining the term "bug" to refer to a hardware glitch, but it was used at Harvard prior to her arrival there. However, it is true that Hopper and her programming team found a moth in one of the relays in the computer hardware that shut the system down. This famous moth was pasted into a lab book. In the 1950s Hopper coined the term "debug" for the process of removing programming errors.

In 1946, when the Navy told her that she was too old for active service, Hopper chose to remain at Harvard as a civilian research fellow. In 1949 she left Harvard to join the Eckert-Mauchly Computer Corporation, where she helped develop the first commercial computer, UNIVAC. Hopper remained with this company when it was taken over by Remington Rand and when Remington Rand merged with the Sperry Corporation. She was a visionary for the potential power of computers; she understood that computers would become widely used if tools that were both programmer-friendly and application-friendly could be developed. In particular, she believed that computer programs could be written in English, rather than using machine instructions. To help achieve this goal, she developed the first compiler. She published the first research paper on compilers in 1952. Hopper is also known as the mother of the computer language COBOL; members of Hopper's staff helped to frame the basic language design for COBOL using their earlier work as a basis.

In 1966, Hopper retired from the Navy Reserve. However, only seven months later, the Navy recalled her from retirement to help standardize high-level naval computer languages. In 1983 she was promoted to the rank of Commodore by special Presidential appointment, and in 1985 she was elevated to the rank of Rear Admiral. Her retirement from the Navy, at the age of 80, was held on the *USS Constitution*.

## Nondeterministic Finite-State Automata

The finite-state automata discussed so far are **deterministic**, because for each pair of state and input value there is a unique next state given by the transition function. There is another important type of finite-state automaton in which there may be several possible next states for each pair of input value and state. Such machines are called **nondeterministic**. Nondeterministic finite-state automata are important in determining which languages can be recognized by a finite-state automaton.

### DEFINITION 5



A *nondeterministic finite-state automaton*  $M = (S, I, f, s_0, F)$  consists of a set  $S$  of states, an input alphabet  $I$ , a transition function  $f$  that assigns a set of states to each pair of state and input (so that  $f : S \times I \rightarrow P(S)$ ), a starting state  $s_0$ , and a subset  $F$  of  $S$  consisting of the final states.

We can represent nondeterministic finite-state automata using state tables or state diagrams. When we use a state table, for each pair of state and input value we give a list of possible next states. In the state diagram, we include an edge from each state to all possible next states, labeling edges with the input or inputs that lead to this transition.

**EXAMPLE 9** Find the state diagram for the nondeterministic finite-state automaton with the state table shown in Table 2. The final states are  $s_2$  and  $s_3$ .

*Solution:* The state diagram for this automaton is shown in Figure 6. ▶

**EXAMPLE 10** Find the state table for the nondeterministic finite-state automaton with the state diagram shown in Figure 7.

*Solution:* The state table is given as Table 3. ▶

What does it mean for a nondeterministic finite-state automaton to recognize a string  $x = x_1x_2 \dots x_k$ ? The first input symbol  $x_1$  takes the starting state  $s_0$  to a set  $S_1$  of states. The next input symbol  $x_2$  takes each of the states in  $S_1$  to a set of states. Let  $S_2$  be the union of these sets. We continue this process, including at a stage all states obtained using a state obtained at the previous stage and the current input symbol. We **recognize**, or **accept**, the string  $x$  if there is a final state in the set of all states that can be obtained from  $s_0$  using  $x$ . The **language recognized** by a nondeterministic finite-state automaton is the set of all strings recognized by this automaton.

TABLE 2

| State | $f$             |            |
|-------|-----------------|------------|
|       | 0               | 1          |
| $s_0$ | $s_0, s_1$      | $s_3$      |
| $s_1$ | $s_0$           | $s_1, s_3$ |
| $s_2$ |                 | $s_0, s_2$ |
| $s_3$ | $s_0, s_1, s_2$ | $s_1$      |

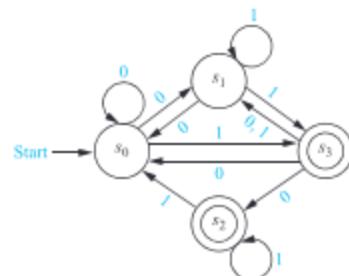
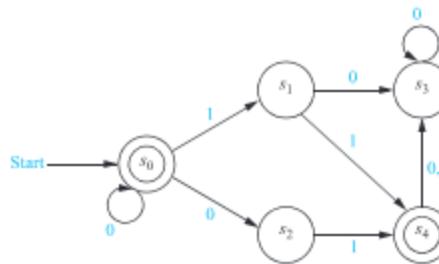


FIGURE 6 The Nondeterministic Finite-State Automaton with State Table Given in Table 2.



**FIGURE 7** A Nondeterministic Finite-State Automaton.

| State | <i>f</i>   |       |
|-------|------------|-------|
|       | Input      |       |
|       | 0          | 1     |
| $s_0$ | $s_0, s_2$ | $s_1$ |
| $s_1$ | $s_3$      | $s_4$ |
| $s_2$ |            | $s_4$ |
| $s_3$ | $s_3$      |       |
| $s_4$ | $s_3$      | $s_3$ |

**EXAMPLE 11** Find the language recognized by the nondeterministic finite-state automaton shown in Figure 7.

**Solution:** Because  $s_0$  is a final state, and there is a transition from  $s_0$  to itself when 0 is the input, the machine recognizes all strings consisting of zero or more consecutive 0s. Furthermore, because  $s_4$  is a final state, any string that has  $s_4$  in the set of states that can be reached from  $s_0$  with this input string is recognized. The only such strings are strings consisting of zero or more consecutive 0s followed by 01 or 11. Because  $s_0$  and  $s_4$  are the only final states, the language recognized by the machine is  $\{0^n, 0^n01, 0^n11 \mid n \geq 0\}$ .  $\blacktriangleleft$

One important fact is that a language recognized by a nondeterministic finite-state automaton is also recognized by a deterministic finite-state automaton. We will take advantage of this fact in Section 13.4 when we will determine which languages are recognized by finite-state automata.

### THEOREM 1

If the language  $L$  is recognized by a nondeterministic finite-state automaton  $M_0$ , then  $L$  is also recognized by a deterministic finite-state automaton  $M_1$ .

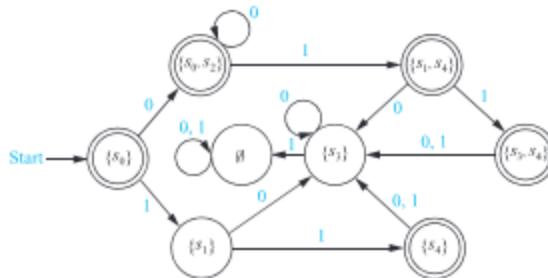


**Proof:** We will describe how to construct the deterministic finite-state automaton  $M_1$  that recognizes  $L$  from  $M_0$ , the nondeterministic finite-state automaton that recognizes this language. Each state in  $M_1$  will be made up of a set of states in  $M_0$ . The start symbol of  $M_1$  is  $\{s_0\}$ , which is the set containing the start state of  $M_0$ . The input set of  $M_1$  is the same as the input set of  $M_0$ .

Given a state  $\{s_{i_1}, s_{i_2}, \dots, s_{i_k}\}$  of  $M_1$ , the input symbol  $x$  takes this state to the union of the sets of next states for the elements of this set, that is, the union of the sets  $f(s_{i_1}, x)$ ,  $f(s_{i_2}, x), \dots, f(s_{i_k}, x)$ . The states of  $M_1$  are all the subsets of  $S$ , the set of states of  $M_0$ , that are obtained in this way starting with  $s_0$ . (There are as many as  $2^n$  states in the deterministic machine, where  $n$  is the number of states in the nondeterministic machine, because all subsets may occur as states, including the empty set, although usually far fewer states occur.) The final states of  $M_1$  are those sets that contain a final state of  $M_0$ .

Suppose that an input string is recognized by  $M_0$ . Then one of the states that can be reached from  $s_0$  using this input string is a final state (the reader should provide an inductive proof of this). This means that in  $M_1$ , this input string leads from  $\{s_0\}$  to a set of states of  $M_0$  that contains a final state. This subset is a final state of  $M_1$ , so this string is also recognized by  $M_1$ . Also, an input string not recognized by  $M_0$  does not lead to any final states in  $M_0$ . (The reader should provide the details that prove this statement.) Consequently, this input string does not lead from  $\{s_0\}$  to a final state in  $M_1$ .  $\blacktriangleleft$

**EXAMPLE 12** Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Example 10.



**FIGURE 8** A Deterministic Automaton Equivalent to the Nondeterministic Automaton in Example 10.

**Solution:** The deterministic automaton shown in Figure 8 is constructed from the nondeterministic automaton in Example 10. The states of this deterministic automaton are subsets of the set of all states of the nondeterministic machine. The next state of a subset under an input symbol is the subset containing the next states in the nondeterministic machine of all elements in this subset. For instance, on input of 0,  $\{s_0\}$  goes to  $\{s_0, s_2\}$ , because  $s_0$  has transitions to itself and to  $s_2$  in the nondeterministic machine; the set  $\{s_0, s_2\}$  goes to  $\{s_1, s_4\}$  on input of 1, because  $s_0$  goes just to  $s_1$  and  $s_2$  goes just to  $s_4$  on input of 1 in the nondeterministic machine; and the set  $\{s_1, s_4\}$  goes to  $\{s_3\}$  on input of 0, because  $s_1$  and  $s_4$  both go to just  $s_3$  on input of 0 in the deterministic machine. All subsets that are obtained in this way are included in the deterministic finite-state machine. Note that the empty set is one of the states of this machine, because it is the subset containing all the next states of  $\{s_3\}$  on input of 1. The start state is  $\{s_0\}$ , and the set of final states are all those that include  $s_0$  or  $s_4$ .  $\blacktriangleleft$

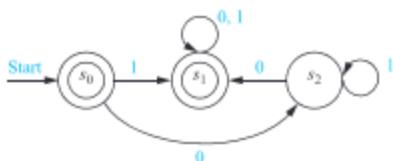
## Exercises

- Let  $A = \{0, 11\}$  and  $B = \{00, 01\}$ . Find each of these sets.  
 a)  $AB$     b)  $BA$     c)  $A^2$     d)  $B^3$
  - Show that if  $A$  is a set of strings, then  $A\emptyset = \emptyset A = \emptyset$ .
  - Find all pairs of sets of strings  $A$  and  $B$  for which  $AB = \{10, 111, 1010, 1000, 10111, 101000\}$ .
  - Show that these equalities hold.  
 a)  $\{\lambda\}^* = \{\lambda\}$   
 b)  $(A^*)^* = A^*$  for every set of strings  $A$
  - Describe the elements of the set  $A^*$  for these values of  $A$ .  
 a)  $\{10\}$     b)  $\{111\}$     c)  $\{0, 01\}$     d)  $\{1, 101\}$
  - Let  $V$  be an alphabet, and let  $A$  and  $B$  be subsets of  $V^*$ . Show that  $|AB| \leq |A||B|$ .
  - Let  $V$  be an alphabet, and let  $A$  and  $B$  be subsets of  $V^*$  with  $A \subseteq B$ . Show that  $A^* \subseteq B^*$ .
  - Suppose that  $A$  is a subset of  $V^*$ , where  $V$  is an alphabet. Prove or disprove each of these statements.  
 a)  $A \subseteq A^2$     b) if  $A = A^2$ , then  $\lambda \in A$   
 c)  $A\{\lambda\} = A$     d)  $(A^*)^* = A^*$   
 e)  $A^*A = A^*$     f)  $|A^n| = |A|^n$
  - Determine whether the string 11101 is in each of these sets.  
 a)  $\{0, 1\}^*$     b)  $\{1\}^*\{0\}^*\{1\}^*$
- $\{11\}\{0\}^*\{01\}$     d)  $\{11\}^*\{01\}^*$   
 e)  $\{111\}^*\{0\}^*\{1\}$     f)  $\{11, 0\}\{00, 101\}$
  - Determine whether the string 01001 is in each of these sets.  
 a)  $\{0, 1\}^*$     b)  $\{0\}^*\{10\}\{1\}^*$   
 c)  $\{010\}^*\{0\}^*\{1\}$     d)  $\{010, 011\}\{00, 01\}$   
 e)  $\{00\}\{0\}^*\{01\}$     f)  $\{01\}^*\{01\}^*$
  - Determine whether each of these strings is recognized by the deterministic finite-state automaton in Figure 1.  
 a) 111    b) 0011    c) 1010111    d) 011011011
  - Determine whether each of these strings is recognized by the deterministic finite-state automaton in Figure 1.  
 a) 010    b) 1101    c) 1111110    d) 010101010
  - Determine whether all the strings in each of these sets are recognized by the deterministic finite-state automaton in Figure 1.  
 a)  $\{0\}^*$     b)  $\{0\}\{0\}^*$     c)  $\{1\}\{0\}^*$   
 d)  $\{01\}^*$     e)  $\{0\}^*\{1\}^*$     f)  $\{1\}\{0, 1\}^*$
  - Show that if  $M = (S, I, f, s_0, F)$  is a deterministic finite-state automaton and  $f(s, x) = s$  for the state  $s \in S$  and the input string  $x \in I^*$ , then  $f(s, x^n) = s$  for every non-negative integer  $n$ . (Here  $x^n$  is the concatenation of  $n$  copies of the string  $x$ , defined recursively in Exercise 37 in Section 5.3.)

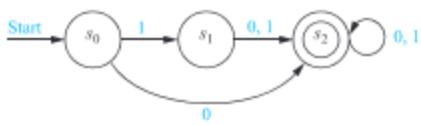
15. Given a deterministic finite-state automaton  $M = (S, I, f, s_0, F)$ , use structural induction and the recursive definition of the extended transition function  $f$  to prove that  $f(s, xy) = f(f(s, x), y)$  for all states  $s \in S$  and all strings  $x \in I^*$  and  $y \in I^*$ .

In Exercises 16–22 find the language recognized by the given deterministic finite-state automaton.

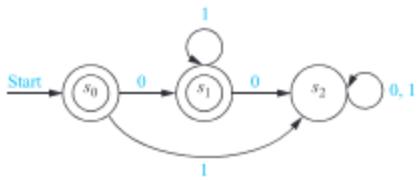
16.



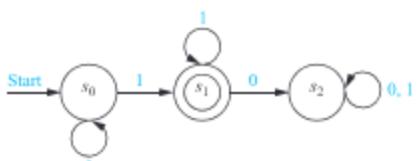
17.



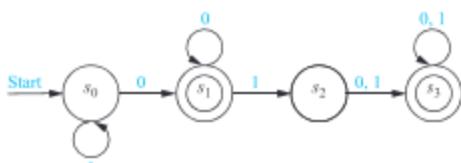
18.



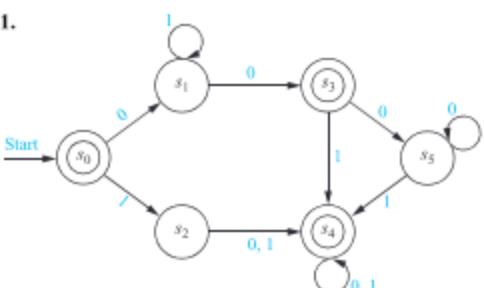
19.



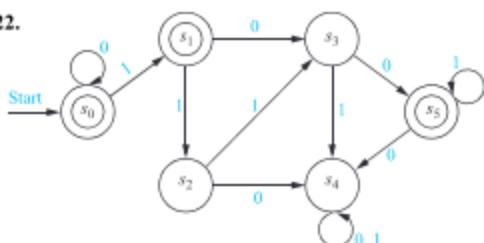
20.



21.



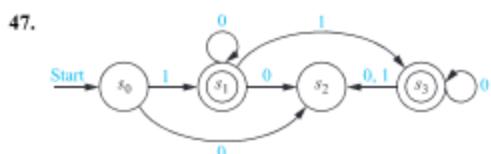
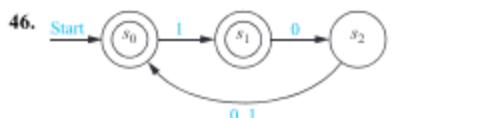
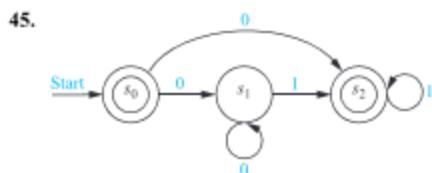
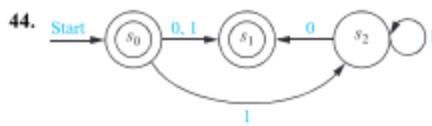
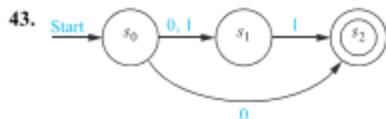
22.



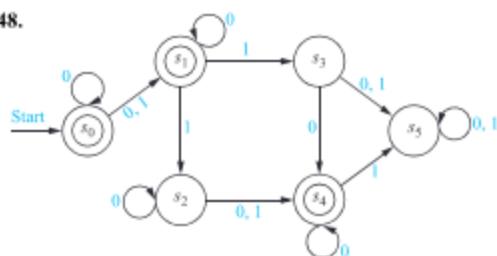
23. Construct a deterministic finite-state automaton that recognizes the set of all bit strings beginning with 01.
24. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that end with 10.
25. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain the string 101.
26. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that do not contain three consecutive 0s.
27. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain exactly three 0s.
28. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain at least three 0s.
29. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain three consecutive 1s.
30. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that begin with 0 or with 11.
31. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that begin and end with 11.
32. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain an even number of 1s.
33. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain an odd number of 0s.
34. Construct a deterministic finite-state automaton that recognizes the set of all bit strings that contain an even number of 0s and an odd number of 1s.
35. Construct a finite-state automaton that recognizes the set of bit strings consisting of a 0 followed by a string with an odd number of 1s.
36. Construct a finite-state automaton with four states that recognizes the set of bit strings containing an even number of 1s and an odd number of 0s.
37. Show that there is no finite-state automaton with two states that recognizes the set of all bit strings that have one or more 1 bits and end with a 0.
38. Show that there is no finite-state automaton with three states that recognizes the set of bit strings containing an even number of 1s and an even number of 0s.

39. Explain how you can change the deterministic finite-state automaton  $M$  so that the changed automaton recognizes the set  $I^* - L(M)$ .
40. Use Exercise 39 and finite-state automata constructed in Example 6 to find deterministic finite-state automata that recognize each of these sets.
- the set of bit strings that do not begin with two 0s
  - the set of bit strings that do not end with two 0s
  - the set of bit strings that contain at most one 0 (that is, that do not contain at least two 0s)
41. Use the procedure you described in Exercise 39 and the finite-state automata you constructed in Exercise 25 to find a deterministic finite-state automaton that recognizes the set of all bit strings that do not contain the string 101.
42. Use the procedure you described in Exercise 39 and the finite-state automaton you constructed in Exercise 29 to find a deterministic finite-state automaton that recognizes the set of all bit strings that do not contain three consecutive 1s.

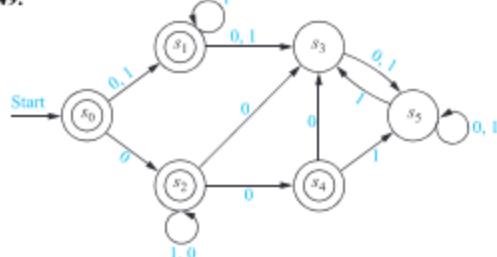
In Exercises 43–49 find the language recognized by the given nondeterministic finite-state automaton.



48.



49.



50. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 43.

51. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 44.

52. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 45.

53. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 46.

54. Find a deterministic finite-state automaton that recognizes the same language as the nondeterministic finite-state automaton in Exercise 47.

55. Find a deterministic finite-state automaton that recognizes each of these sets.

- $\{0\}$
- $\{1, 00\}$
- $\{1^n \mid n = 2, 3, 4, \dots\}$

56. Find a nondeterministic finite-state automaton that recognizes each of the languages in Exercise 55, and has fewer states, if possible, than the deterministic automaton you found in that exercise.

- \*57. Show that there is no finite-state automaton that recognizes the set of bit strings containing an equal number of 0s and 1s.

In Exercises 58–62 we introduce a technique for constructing a deterministic finite-state machine equivalent to a given deterministic finite-state machine with the least number of states possible. Suppose that  $M = (S, I, f, s_0, F)$  is a finite-state automaton and that  $k$  is a nonnegative integer. Let  $R_k$  be the relation on the set  $S$  of states of  $M$  such that  $s R_k t$  if and only if for every input string  $x$  with  $l(x) \leq k$  [where  $l(x)$  is the length of  $x$ , as usual],  $f(s, x) = f(t, x)$  and  $f(t, x)$  is both final states

or both not final states. Furthermore, let  $R_*$  be the relation on the set of states of  $M$  such that  $s R_* t$  if and only if for every input string  $x$ , regardless of length,  $f(s, x)$  and  $f(t, x)$  are both final states or both not final states.

- \*58. a) Show that for every nonnegative integer  $k$ ,  $R_k$  is an equivalence relation on  $S$ . We say that two states  $s$  and  $t$  are  **$k$ -equivalent** if  $s R_k t$ .
  - b) Show that  $R_*$  is an equivalence relation on  $S$ . We say that two states  $s$  and  $t$  are  **$*$ -equivalent** if  $s R_* t$ .
  - c) Show that if  $s$  and  $t$  are two  $k$ -equivalent states of  $M$ , where  $k$  is a positive integer, then  $s$  and  $t$  are also  $(k - 1)$ -equivalent.
  - d) Show that the equivalence classes of  $R_k$  are a refinement of the equivalence classes of  $R_{k-1}$  if  $k$  is a positive integer. (The refinement of a partition of a set is defined in the preamble to Exercise 49 in Section 9.5.)
  - e) Show that if  $s$  and  $t$  are  $k$ -equivalent for every nonnegative integer  $k$ , then they are  $*$ -equivalent.
  - f) Show that all states in a given  $R_*$ -equivalence class are final states or all are not final states.
  - g) Show that if  $s$  and  $t$  are  $R_*$ -equivalent, then  $f(s, a)$  and  $f(t, a)$  are also  $R_*$ -equivalent for all  $a \in I$ .
- \*59. Show that there is a nonnegative integer  $n$  such that the set of  $n$ -equivalence classes of states of  $M$  is the same as the set of  $(n + 1)$ -equivalence classes of states of  $M$ . Then show for this integer  $n$ , the set of  $n$ -equivalence classes of states of  $M$  equals the set of  $*$ -equivalence classes of states of  $M$ .

The **quotient automaton**  $\overline{M}$  of the deterministic finite-state automaton  $M = (S, I, f, s_0, F)$  is the finite-state automaton  $(\overline{S}, I, \overline{f}, [s_0]_{R_*}, \overline{F})$ , where the set of states  $\overline{S}$  is the set of  $*$ -equivalence classes of  $S$ , the transition function  $\overline{f}$  is defined by  $\overline{f}([s]_{R_*}, a) = [f(s, a)]_{R_*}$  for all states  $[s]_{R_*}$  of  $\overline{M}$  and input symbols  $a \in I$ , and  $\overline{F}$  is the set consisting of  $R_*$ -equivalence classes of final states of  $M$ .

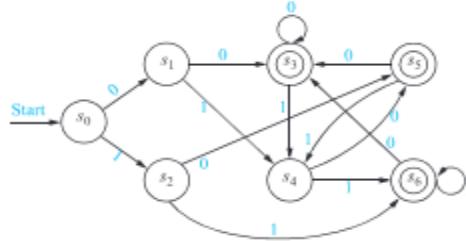
- \*60. a) Show that  $s$  and  $t$  are 0-equivalent if and only if either both  $s$  and  $t$  are final states or neither  $s$  nor  $t$  is a final state. Conclude that each final state of  $\overline{M}$ , which is an  $R_*$ -equivalence class, contains only final states of  $M$ .

- b) Show that if  $k$  is a positive integer, then  $s$  and  $t$  are  $k$ -equivalent if and only if  $s$  and  $t$  are  $(k - 1)$ -equivalent and for every input symbol  $a \in I$ ,  $f(s, a)$  and  $f(t, a)$  are  $(k - 1)$ -equivalent. Conclude that the transition function  $\overline{f}$  is well-defined.
- c) Describe a procedure that can be used to construct the quotient automaton of a finite-automaton  $M$ .

- \*\*61. a) Show that if  $M$  is a finite-state automaton, then the quotient automaton  $\overline{M}$  recognizes the same language as  $M$ .

- b) Show that if  $M$  is a finite-state automaton with the property that for every state  $s$  of  $M$  there is a string  $x \in I^*$  such that  $f(s_0, x) = s$ , then the quotient automaton  $\overline{M}$  has the minimum number of states of any finite-state automaton equivalent to  $M$ .

62. Answer these questions about the finite-state automaton  $M$  shown here.



- a) Find the  $k$ -equivalence classes of  $M$  for  $k = 0, 1, 2$ , and 3. Also, find the  $*$ -equivalence classes of  $M$ .
- b) Construct the quotient automaton  $\overline{M}$  of  $M$ .

## 13.4 Language Recognition

### Introduction

We have seen that finite-state automata can be used as language recognizers. What sets can be recognized by these machines? Although this seems like an extremely difficult problem, there is a simple characterization of the sets that can be recognized by finite state automata. This problem was first solved in 1956 by the American mathematician Stephen Kleene. He showed that there is a finite-state automaton that recognizes a set if and only if this set can be built up from the null set, the empty string, and singleton strings by taking concatenations, unions, and Kleene closures, in arbitrary order. Sets that can be built up in this way are called **regular sets**. Regular grammars were defined in Section 13.1. Because of the terminology used, it is not surprising that there is a connection between regular sets, which are the sets recognized by finite-state automata, and regular grammars. In particular, a set is regular if and only if it is generated by a regular grammar.

Finally, there are sets that cannot be recognized by any finite-state automata. We will give an example of such a set. We will briefly discuss more powerful models of computation, such as pushdown automata and Turing machines, at the end of this section. The regular sets are those

that can be formed using the operations of concatenation, union, and Kleene closure in arbitrary order, starting with the empty set, the set consisting of the empty string, and singleton sets. We will see that the regular sets are those that can be recognized using a finite-state automaton. To define regular sets we first need to define regular expressions.

### DEFINITION 1

The *regular expressions* over a set  $I$  are defined recursively by:

- the symbol  $\emptyset$  is a regular expression;
- the symbol  $\lambda$  is a regular expression;
- the symbol  $x$  is a regular expression whenever  $x \in I$ ;
- the symbols  $(AB)$ ,  $(A \cup B)$ , and  $A^*$  are regular expressions whenever  $A$  and  $B$  are regular expressions.

Each regular expression represents a set specified by these rules:

- $\emptyset$  represents the empty set, that is, the set with no strings;
- $\lambda$  represents the set  $\{\lambda\}$ , which is the set containing the empty string;
- $x$  represents the set  $\{x\}$  containing the string with one symbol  $x$ ;
- $(AB)$  represents the concatenation of the sets represented by  $A$  and by  $B$ ;
- $(A \cup B)$  represents the union of the sets represented by  $A$  and by  $B$ ;
- $A^*$  represents the Kleene closure of the set represented by  $A$ .

Sets represented by regular expressions are called **regular sets**. Henceforth regular expressions will be used to describe regular sets, so when we refer to the regular set  $A$ , we will mean the regular set represented by the regular expression  $A$ . Note that we will leave out outer parentheses from regular expressions when they are not needed.

Example 1 shows how regular expressions are used to specify regular sets.

**EXAMPLE 1** What are the strings in the regular sets specified by the regular expressions  $10^*$ ,  $(10)^*$ ,  $0 \cup 01$ ,  $0(0 \cup 1)^*$ , and  $(0^*1)^*$ ?

*Solution:* The regular sets represented by these expressions are given in Table 1, as the reader should verify. ▲

Finding a regular expression that specifies a given set can be quite tricky, as Example 2 illustrates.

**EXAMPLE 2** Find a regular expression that specifies each of these sets:

- the set of bit strings with even length
- the set of bit strings ending with a 0 and not containing 11
- the set of bit strings containing an odd number of 0s

TABLE 1

| Expression      | Strings                                                |
|-----------------|--------------------------------------------------------|
| $10^*$          | a 1 followed by any number of 0s (including no zeros)  |
| $(10)^*$        | any number of copies of 10 (including the null string) |
| $0 \cup 01$     | the string 0 or the string 01                          |
| $0(0 \cup 1)^*$ | any string beginning with 0                            |
| $(0^*1)^*$      | any string not ending with 0                           |

**Solution:** (a) To construct a regular expression for the set of bit strings with even length, we use the fact that such a string can be obtained by concatenating zero or more strings each consisting of two bits. The set of strings of two bits is specified by the regular expression  $(00 \cup 01 \cup 10 \cup 11)$ . Consequently, the set of strings with even length is specified by  $(00 \cup 01 \cup 10 \cup 11)^*$ .

(b) A bit string ending with a 0 and not containing 11 must be the concatenation of one or more strings where each string is either a 0 or a 10. (To see this, note that such a bit string must consist of 0 bits or 1 bits each followed by a 0; the string cannot end with a single 1 because we know it ends with a 0.) It follows that the regular expression  $(0 \cup 10)^*(0 \cup 10)$  specifies the set of bit strings that do not contain 11 and end with a 0. [Note that the set specified by  $(0 \cup 10)^*$  includes the empty string, which is not in this set, because the empty string does not end with a 0.]

(c) A bit string containing an odd number of 0s must contain at least one 0, which tells us that it starts with zero or more 1s, followed by a 0, followed by zero or more 1s. That is, each such bit string begins with a string of the form  $1^j 0 1^k$  for nonnegative integers  $j$  and  $k$ . Because the bit string contains an odd number of 0s, additional bits after this initial block can be split into blocks each starting with a 0 and containing one more 0. Each such block is of the form  $01^p 0 1^q$ , where  $p$  and  $q$  are nonnegative integers. Consequently, the regular expression  $1^* 0 1^* (01^* 0 1^*)^*$  specifies the set of bit strings with an odd number of 0s. 

## Kleene's Theorem

In 1956 Kleene proved that regular sets are the sets that are recognized by a finite-state automaton. Consequently, this important result is called Kleene's theorem.

### THEOREM 1

**KLEENE'S THEOREM** A set is regular if and only if it is recognized by a finite-state automaton.



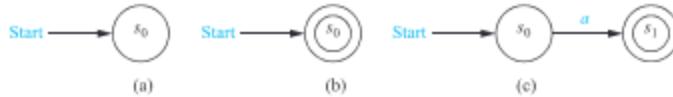
Kleene's theorem is one of the central results in automata theory. We will prove the *only if* part of this theorem, namely, that every regular set is recognized by a finite-state automaton. The proof of the *if* part, that a set recognized by a finite-state automaton is regular, is left as an exercise for the reader.

**Proof:** Recall that a regular set is defined in terms of regular expressions, which are defined recursively. We can prove that every regular set is recognized by a finite-state automaton if we can do the following things.



1. Show that  $\emptyset$  is recognized by a finite-state automaton.
2. Show that  $\{\lambda\}$  is recognized by a finite-state automaton.
3. Show that  $\{a\}$  is recognized by a finite-state automaton whenever  $a$  is a symbol in  $I$ .
4. Show that  $AB$  is recognized by a finite-state automaton whenever both  $A$  and  $B$  are.
5. Show that  $A \cup B$  is recognized by a finite-state automaton whenever both  $A$  and  $B$  are.
6. Show that  $A^*$  is recognized by a finite-state automaton whenever  $A$  is.

We now consider each of these tasks. First, we show that  $\emptyset$  is recognized by a nondeterministic finite-state automaton. To do this, all we need is an automaton with no final states. Such an automaton is shown in Figure 1(a).



**FIGURE 1** Nondeterministic Finite-State Automata That Recognize Some Basic Sets.

Second, we show that  $\{\lambda\}$  is recognized by a finite-state automaton. To do this, all we need is an automaton that recognizes  $\lambda$ , the null string, but not any other string. This can be done by making the start state  $s_0$  a final state and having no transitions, so that no other string takes  $s_0$  to a final state. The nondeterministic automaton in Figure 1(b) shows such a machine.

Third, we show that  $\{a\}$  is recognized by a nondeterministic finite-state automaton. To do this, we can use a machine with a starting state  $s_0$  and a final state  $s_1$ . We have a transition from  $s_0$  to  $s_1$  when the input is  $a$ , and no other transitions. The only string recognized by this machine is  $a$ . This machine is shown in Figure 1(c).

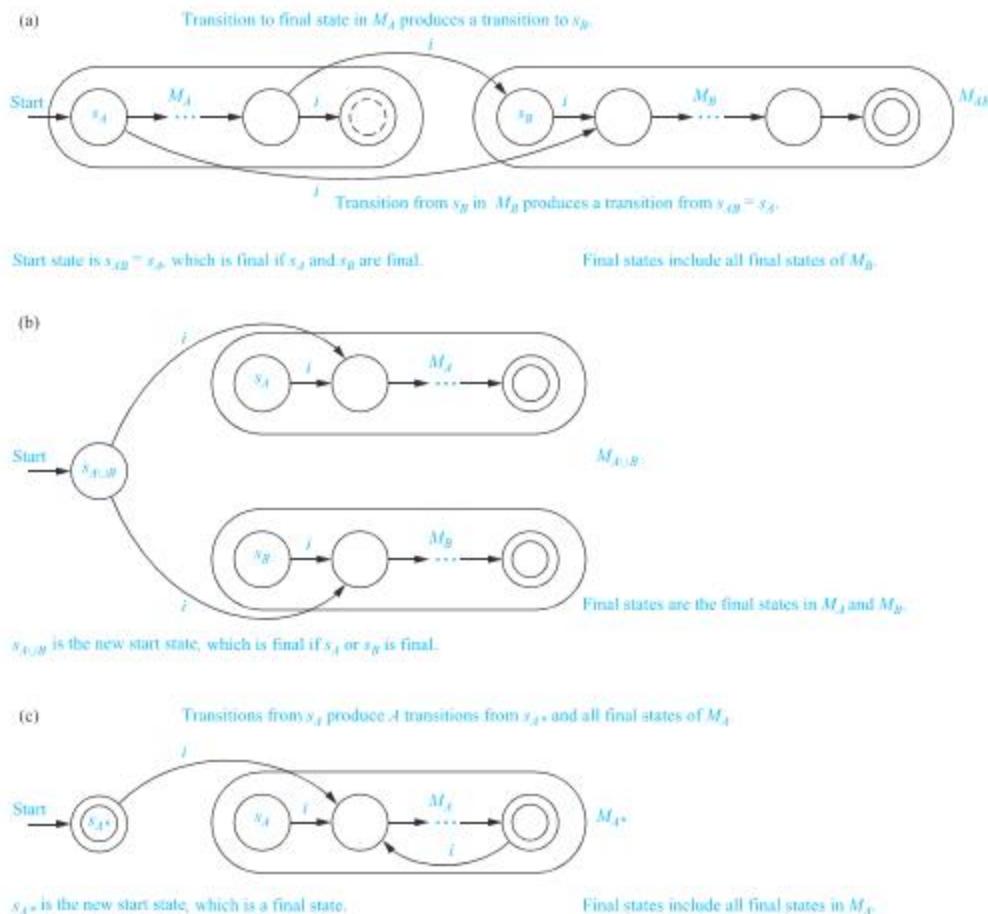
Next, we show that  $AB$  and  $A \cup B$  can be recognized by finite-state automata if  $A$  and  $B$  are languages recognized by finite-state automata. Suppose that  $A$  is recognized by  $M_A = (S_A, I, f_A, s_A, F_A)$  and  $B$  is recognized by  $M_B = (S_B, I, f_B, s_B, F_B)$ .

We begin by constructing a finite-state machine  $M_{AB} = (S_{AB}, I, f_{AB}, s_{AB}, F_{AB})$  that recognizes  $AB$ , the concatenation of  $A$  and  $B$ . We build such a machine by combining the machines for  $A$  and  $B$  in series, so a string in  $A$  takes the combined machine from  $s_A$ , the start state of  $M_A$ , to  $s_B$ , the start state of  $M_B$ . A string in  $B$  should take the combined machine from  $s_B$  to a final state of the combined machine. Consequently, we make the following construction. Let  $S_{AB}$  be  $S_A \cup S_B$ . [Note that we can assume that  $S_A$  and  $S_B$  are disjoint.] The starting state  $s_{AB}$  is the same as  $s_A$ . The set of final states,  $F_{AB}$ , is the set of final states of  $M_B$  with  $s_{AB}$  included if and only if  $\lambda \in A \cap B$ . The transitions in  $M_{AB}$  include all transitions in  $M_A$  and in  $M_B$ , as well as some new transitions. For every transition in  $M_A$  that leads to a final state, we form a transition in  $M_{AB}$  from the same state to  $s_B$ , on the same input. In this way, a string in  $A$  takes  $M_{AB}$  from  $s_{AB}$  to  $s_B$ , and then a string in  $B$  takes  $s_B$  to a final state of  $M_{AB}$ . Moreover, for every transition from  $s_B$  we form a transition in  $M_{AB}$  from  $s_{AB}$  to the same state. Figure 2(a) contains an illustration of this construction.

We now construct a machine  $M_{A \cup B} = (S_{A \cup B}, I, f_{A \cup B}, s_{A \cup B}, F_{A \cup B})$  that recognizes  $A \cup B$ . This automaton can be constructed by combining  $M_A$  and  $M_B$  in parallel, using a new start state that has the transitions that both  $s_A$  and  $s_B$  have. Let  $S_{A \cup B} = S_A \cup S_B \cup \{s_{A \cup B}\}$ , where  $s_{A \cup B}$  is a new state that is the start state of  $M_{A \cup B}$ . Let the set of final states  $F_{A \cup B}$  be  $F_A \cup F_B \cup \{s_{A \cup B}\}$  if  $\lambda \in A \cup B$ , and  $F_A \cup F_B$  otherwise. The transitions in  $M_{A \cup B}$  include all those in  $M_A$  and in  $M_B$ . Also, for each transition from  $s_A$  to a state  $s$  on input  $i$  we include a transition from  $s_{A \cup B}$  to  $s$  on input  $i$ , and for each transition from  $s_B$  to a state  $s$  on input  $i$  we include a transition from  $s_{A \cup B}$  to  $s$  on input  $i$ . In this way, a string in  $A$  leads from  $s_{A \cup B}$  to a final state in the new machine, and a string in  $B$  leads from  $s_{A \cup B}$  to a final state in the new machine. Figure 2(b) illustrates the construction of  $M_{A \cup B}$ .

Finally, we construct  $M_{A^*} = (S_{A^*}, I, f_{A^*}, s_{A^*}, F_{A^*})$ , a machine that recognizes  $A^*$ , the Kleene closure of  $A$ . Let  $S_{A^*}$  include all states in  $S_A$  and one additional state  $s_{A^*}$ , which is the starting state for the new machine. The set of final states  $F_{A^*}$  includes all states in  $F_A$  as well as the start state  $s_{A^*}$ , because  $\lambda$  must be recognized. To recognize concatenations of arbitrarily many strings from  $A$ , we include all the transitions in  $M_A$ , as well as transitions from  $s_{A^*}$  that match the transitions from  $s_A$ , and transitions from each final state that match the transitions from  $s_A$ . With this set of transitions, a string made up of concatenations of strings from  $A$  will take  $s_{A^*}$  to a final state when the first string in  $A$  has been read, returning to a final state when the second string in  $A$  has been read, and so on. Figure 2(c) illustrates the construction we used.  $\triangle$

A nondeterministic finite-state automaton can be constructed for any regular set using the procedure described in this proof. We illustrate how this is done with Example 3.

**FIGURE 2** Building Automata to Recognize Concatenations, Unions, and Kleene Closures.

**EXAMPLE 3** Construct a nondeterministic finite-state automaton that recognizes the regular set  $1^* \cup 01$ .

**Solution:** We begin by building a machine that recognizes  $1^*$ . This is done using the machine that recognizes  $1$  and then using the construction for  $M_A^*$  described in the proof. Next, we build a machine that recognizes  $01$ , using machines that recognize  $0$  and  $1$  and the construction in the proof for  $M_{AB}$ . Finally, using the construction in the proof for  $M_{A \cup B}$ , we construct the machine for  $1^* \cup 01$ . The finite-state automata used in this construction are shown in Figure 3. The states in the successive machines have been labeled using different subscripts, even when a state is formed from one previously used in another machine. Note that the construction given here does not produce the simplest machine that recognizes  $1^* \cup 01$ . A much simpler machine that recognizes this set is shown in Figure 3(b).  $\blacktriangleleft$

## Regular Sets and Regular Grammars

In Section 13.1 we introduced phrase-structure grammars and defined different types of grammars. In particular we defined regular, or type 3, grammars, which are grammars of the

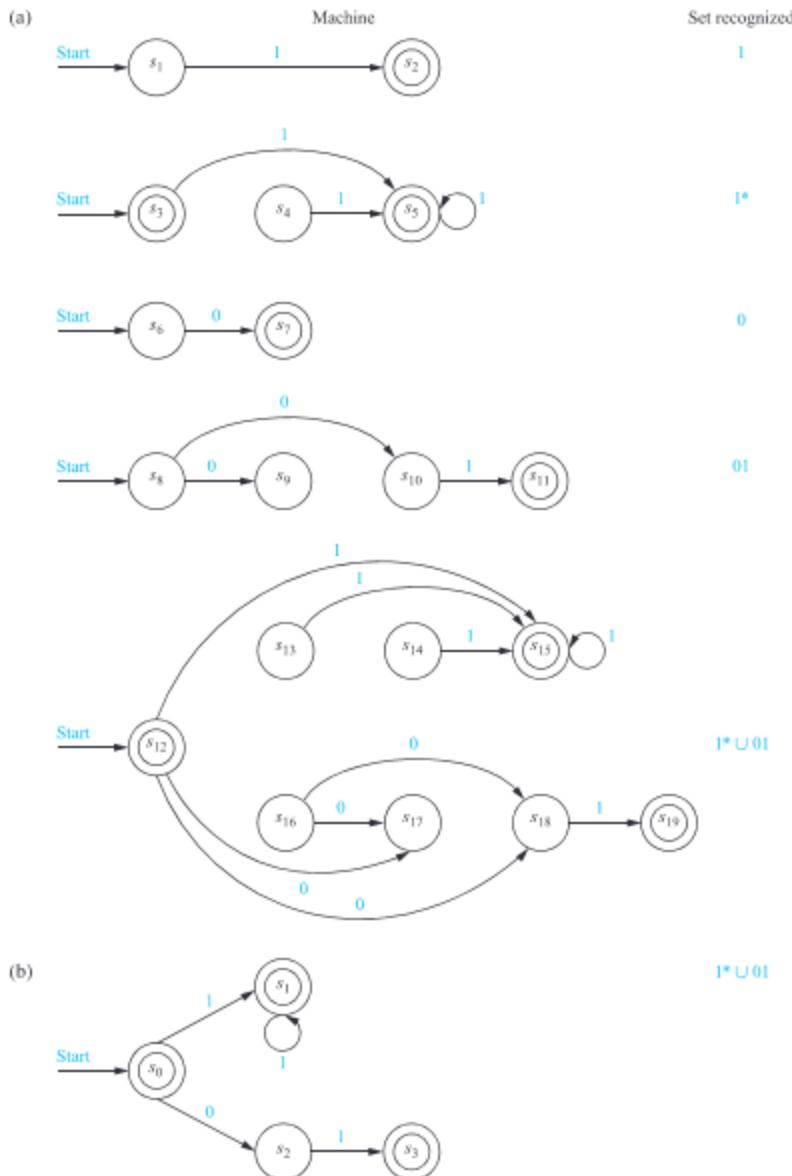
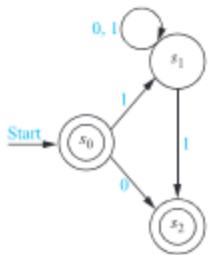


FIGURE 3 Nondeterministic Finite-State Automata Recognizing  $1^* \cup 01$ .

form  $G = (V, T, S, P)$ , where each production is of the form  $S \rightarrow \lambda$ ,  $A \rightarrow a$ , or  $A \rightarrow aB$ , where  $a$  is a terminal symbol, and  $A$  and  $B$  are nonterminal symbols. As the terminology suggests, there is a close connection between regular grammars and regular sets.

#### THEOREM 2

A set is generated by a regular grammar if and only if it is a regular set.



**FIGURE 4** A Nondeterministic Finite-State Automaton Recognizing  $L(G)$ .

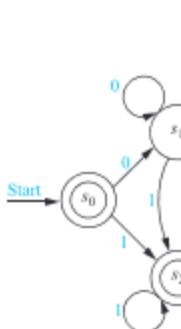
**Proof:** First we show that a set generated by a regular grammar is a regular set. Suppose that  $G = (V, T, S, P)$  is a regular grammar generating the set  $L(G)$ . To show that  $L(G)$  is regular we will build a nondeterministic finite-state machine  $M = (S, I, f, s_0, F)$  that recognizes  $L(G)$ . Let  $S$ , the set of states, contain a state  $s_A$  for each nonterminal symbol  $A$  of  $G$  and an additional state  $s_F$ , which is a final state. The start state  $s_0$  is the state formed from the start symbol  $S$ . The transitions of  $M$  are formed from the productions of  $G$  in the following way. A transition from  $s_A$  to  $s_F$  on input of  $a$  is included if  $A \rightarrow a$  is a production, and a transition from  $s_A$  to  $s_B$  on input of  $a$  is included if  $A \rightarrow aB$  is a production. The set of final states includes  $s_F$  and also includes  $s_0$  if  $S \rightarrow \lambda$  is a production in  $G$ . It is not hard to show that the language recognized by  $M$  equals the language generated by the grammar  $G$ , that is,  $L(M) = L(G)$ . This can be done by determining the words that lead to a final state. The details are left as an exercise for the reader.  $\triangleleft$

Before giving the proof of the converse, we illustrate how a nondeterministic machine is constructed that recognizes the same set as a regular grammar.

**EXAMPLE 4** Construct a nondeterministic finite-state automaton that recognizes the language generated by the regular grammar  $G = (V, T, S, P)$ , where  $V = \{0, 1, A, S\}$ ,  $T = \{0, 1\}$ , and the productions in  $P$  are  $S \rightarrow 1A$ ,  $S \rightarrow 0$ ,  $S \rightarrow \lambda$ ,  $A \rightarrow 0A$ ,  $A \rightarrow 1A$ , and  $A \rightarrow 1$ .

**Solution:** The state diagram for a nondeterministic finite-state automaton that recognizes  $L(G)$  is shown in Figure 4. This automaton is constructed following the procedure described in the proof. In this automaton,  $s_0$  is the state corresponding to  $S$ ,  $s_1$  is the state corresponding to  $A$ , and  $s_2$  is the final state.  $\triangleleft$

We now complete the proof of Theorem 2.



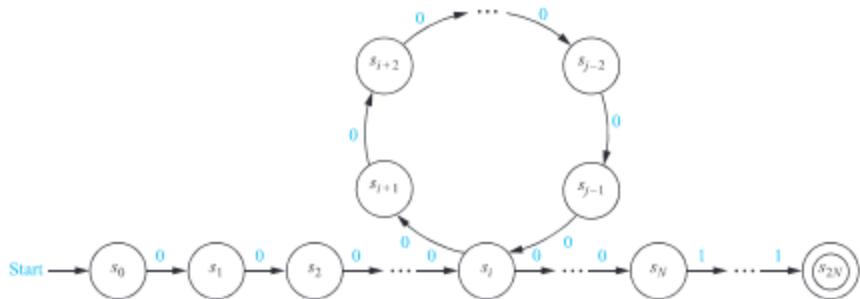
**FIGURE 5** A Finite-State Automaton.

**Proof:** We now show that if a set is regular, then there is a regular grammar that generates it. Suppose that  $M$  is a finite-state machine that recognizes this set with the property that  $s_0$ , the starting state of  $M$ , is never the next state for a transition. (We can find such a machine by Exercise 20.) The grammar  $G = (V, T, S, P)$  is defined as follows. The set  $V$  of symbols of  $G$  is formed by assigning a symbol to each state of  $S$  and each input symbol in  $I$ . The set  $T$  of terminal symbols of  $G$  is the set  $I$ . The start symbol  $S$  is the symbol formed from the start state  $s_0$ . The set  $P$  of productions in  $G$  is formed from the transitions in  $M$ . In particular, if the state  $s$  goes to a final state under input  $a$ , then the production  $A_s \rightarrow a$  is included in  $P$ , where  $A_s$  is the nonterminal symbol formed from the state  $s$ . If the state  $s$  goes to the state  $t$  on input  $a$ , then the production  $A_s \rightarrow aA_t$  is included in  $P$ . The production  $S \rightarrow \lambda$  is included in  $P$  if and only if  $\lambda \in L(M)$ . Because the productions of  $G$  correspond to the transitions of  $M$  and the productions leading to terminals correspond to transitions to final states, it is not hard to show that  $L(G) = L(M)$ . We leave the details as an exercise for the reader.  $\triangleleft$

Example 5 illustrates the construction used to produce a grammar from an automaton that generates the language recognized by this automaton.

**EXAMPLE 5** Find a regular grammar that generates the regular set recognized by the finite-state automaton shown in Figure 5.

**Solution:** The grammar  $G = (V, T, S, P)$  generates the set recognized by this automaton where  $V = \{S, A, B, 0, 1\}$ , the symbols  $S$ ,  $A$ , and  $B$  correspond to the states  $s_0$ ,  $s_1$ , and  $s_2$ , respectively,  $T = \{0, 1\}$ ,  $S$  is the start symbol; and the productions are  $S \rightarrow 0A$ ,  $S \rightarrow 1B$ ,  $S \rightarrow 1$ ,  $S \rightarrow \lambda$ ,  $A \rightarrow 0A$ ,  $A \rightarrow 1B$ ,  $A \rightarrow 1$ ,  $B \rightarrow 0A$ ,  $B \rightarrow 1B$ , and  $B \rightarrow 1$ .  $\triangleleft$



**FIGURE 6** The Path Produced by  $0^N 1^N$ .

### A Set Not Recognized by a Finite-State Automaton

We have seen that a set is recognized by a finite state automaton if and only if it is regular. We will now show that there are sets that are not regular by describing one such set. The technique used to show that this set is not regular illustrates an important method for showing that certain sets are not regular.

**EXAMPLE 6** Show that the set  $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$ , made up of all strings consisting of a block of 0s followed by a block of an equal number of 1s, is not regular.

**Solution:** Suppose that this set were regular. Then there would be a nondeterministic finite-state automaton  $M = (S, I, f, s_0, F)$  recognizing it. Let  $N$  be the number of states in this machine, that is,  $N = |S|$ . Because  $M$  recognizes all strings made up of a number of 0s followed by an equal number of 1s,  $M$  must recognize  $0^N 1^N$ . Let  $s_0, s_1, s_2, \dots, s_{2N}$  be the sequence of states that is obtained starting at  $s_0$  and using the symbols of  $0^N 1^N$  as input, so that  $s_1 = f(s_0, 0)$ ,  $s_2 = f(s_1, 0), \dots, s_N = f(s_{N-1}, 0)$ ,  $s_{N+1} = f(s_N, 1), \dots, s_{2N} = f(s_{2N-1}, 1)$ . Note that  $s_{2N}$  is a final state.

Because there are only  $N$  states, the pigeonhole principle shows that at least two of the first  $N + 1$  of the states, which are  $s_0, \dots, s_N$ , must be the same. Say that  $s_i$  and  $s_j$  are two such identical states, with  $0 \leq i < j \leq N$ . This means that  $f(s_i, 0^t) = s_j$ , where  $t = j - i$ . It follows that there is a loop leading from  $s_i$  back to itself, obtained using the input 0 a total of  $t$  times, in the state diagram shown in Figure 6.

Now consider the input string  $0^N 0^t 1^N = 0^{N+t} 1^N$ . There are  $t$  more consecutive 0s at the start of this block than there are consecutive 1s that follow it. Because this string is not of the form  $0^n 1^n$  (because it has more 0s than 1s), it is not recognized by  $M$ . Consequently,  $f(s_0, 0^{N+t} 1^N)$  cannot be a final state. However, when we use the string  $0^{N+t} 1^N$  as input, we end up in the same state as before, namely,  $s_{2N}$ . The reason for this is that the extra  $t$  0s in this string take us around the loop from  $s_i$  back to itself an extra time, as shown in Figure 6. Then the rest of the string leads us to exactly the same state as before. This contradiction shows that  $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$  is not regular. ▲

### More Powerful Types of Machines



Finite-state automata are unable to carry out many computations. The main limitation of these machines is their finite amount of memory. This prevents them from recognizing languages that are not regular, such as  $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$ . Because a set is regular if and only if it is the language generated by a regular grammar, Example 6 shows that there is no regular grammar

that generates the set  $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$ . However, there is a context-free grammar that generates this set. Such a grammar was given in Example 5 in Section 13.1.

Because of the limitations of finite-state machines, it is necessary to use other, more powerful, models of computation. One such model is the **pushdown automaton**. A pushdown automaton includes everything in a finite-state automaton, as well as a stack, which provides unlimited memory. Symbols can be placed on the top or taken off the top of the stack. A set is recognized in one of two ways by a pushdown automaton. First, a set is recognized if the set consists of all the strings that produce an empty stack when they are used as input. Second, a set is recognized if it consists of all the strings that lead to a final state when used as input. It can be shown that a set is recognized by a pushdown automaton if and only if it is the language generated by a context-free grammar.

However, there are sets that cannot be expressed as the language generated by a context-free grammar. One such set is  $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$ . We will indicate why this set cannot be recognized by a pushdown automaton, but we will not give a proof, because we have not developed the machinery needed. (However, one method of proof is given in Exercise 28 of the supplementary exercises at the end of this chapter.) The stack can be used to show that a string begins with a sequence of 0s followed by an equal number of 1s by placing a symbol on the stack for each 0 (as long as only 0s are read), and removing one of these symbols for each 1 (as long as only 1s following the 0s are read). But once this is done, the stack is empty, and there is no way to determine that there are the same number of 2s in the string as 0s.

There are other machines called **linear bounded automata**, more powerful than pushdown automata, that can recognize sets such as  $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$ . In particular, linear bounded automata can recognize context-sensitive languages. However, these machines cannot recognize all the languages generated by phrase-structure grammars. To avoid the limitations of special types of machines, the model known as a **Turing machine**, named after the British mathematician Alan Turing, is used. A Turing machine is made up of everything included in a finite-state machine together with a tape, which is infinite in both directions. A Turing machine has read and write capabilities on the tape, and it can move back and forth along this tape. Turing machines can recognize all languages generated by phrase-structure grammars. In addition, Turing machines can model all the computations that can be performed on a computing machine. Because of their power, Turing machines are extensively studied in theoretical computer science. We will briefly study them in Section 13.5.

**Alan Turing invented  
Turning machines before  
modern computers  
existed!**

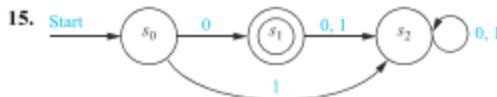
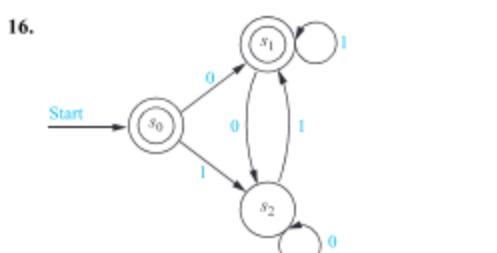
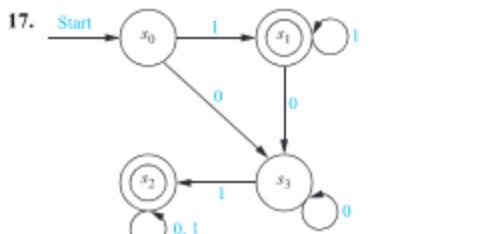


**ALAN MATHISON TURING (1912–1954)** Alan Turing was born in London, although he was conceived in India, where his father was employed in the Indian Civil Service. As a boy, he was fascinated by chemistry, performing a wide variety of experiments, and by machinery. Turing attended Sherborne, an English boarding school. In 1931 he won a scholarship to King's College, Cambridge. After completing his dissertation, which included a rediscovery of the central limit theorem, a famous theorem in statistics, he was elected a fellow of his college. In 1935 Turing became fascinated with the decision problem, a problem posed by the great German mathematician Hilbert, which asked whether there is a general method that can be applied to any assertion to determine whether the assertion is true. Turing enjoyed running (later in life running as a serious amateur in competitions), and one day, while resting after a run, he discovered the key ideas needed to solve the decision problem. In his solution, he invented what is now called a **Turing machine** as the most general model of a computing machine. Using these machines, he found a problem, involving what he called computable numbers, that could not be decided using a general method.

From 1936 to 1938 Turing visited Princeton University to work with Alonzo Church, who had also solved Hilbert's decision problem. In 1939 Turing returned to King's College. However, at the outbreak of World War II, he joined the Foreign Office, performing cryptanalysis of German ciphers. His contribution to the breaking of the code of the Enigma, a mechanical German cipher machine, played an important role in winning the war.

After the war, Turing worked on the development of early computers. He was interested in the ability of machines to think, proposing that if a computer could not be distinguished from a person based on written replies to questions, it should be considered to be "thinking." He was also interested in biology, having written on morphogenesis, the development of form in organisms. In 1954 Turing committed suicide by taking cyanide, without leaving a clear explanation. Legal troubles related to a homosexual relationship and hormonal treatments mandated by the court to lessen his sex drive may have been factors in his decision to end his life.

## Exercises

1. Describe in words the strings in each of these regular sets.
    - a)  $1^*0$
    - b)  $1^*00^*$
    - c)  $111 \cup 001$
    - d)  $(1 \cup 00)^*$
    - e)  $(00^*1)^*$
    - f)  $(0^*1)(0 \cup 1)^*00$
  2. Describe in words the strings in each of these regular sets.
    - a)  $001^*$
    - b)  $(01)^*$
    - c)  $01 \cup 001^*$
    - d)  $0(11 \cup 0)^*$
    - e)  $(101^*)^*$
    - f)  $(0^* \cup 1)11$
  3. Determine whether 0101 belongs to each of these regular sets.
    - a)  $01^*0^*$
    - b)  $0(11)^*(01)^*$
    - c)  $0(10)^*1^*$
    - d)  $0^*10(0 \cup 1)$
    - e)  $(01)^*(11)^*$
    - f)  $0^*(10 \cup 11)^*$
    - g)  $0^*(10)^*11$
    - h)  $01(01 \cup 0)^*1^*$
  4. Determine whether 1011 belongs to each of these regular sets.
    - a)  $10^*1^*$
    - b)  $0^*(10 \cup 11)^*$
    - c)  $1(01)^*1^*$
    - d)  $1^*01(0 \cup 1)$
    - e)  $(10)^*(11)^*$
    - f)  $1(00)^*(11)^*$
    - g)  $(10)^*1011$
    - h)  $(1 \cup 00)(01 \cup 0)^*1^*$
  5. Express each of these sets using a regular expression.
    - a) the set consisting of the strings 0, 11, and 010
    - b) the set of strings of three 0s followed by two or more 0s
    - c) the set of strings of odd length
    - d) the set of strings containing exactly one 1
    - e) the set of strings ending in 1 and not containing 000
  6. Express each of these sets using a regular expression.
    - a) the set containing all strings with zero, one, or two bits
    - b) the set of strings of two 0s, followed by zero or more 1s, and ending with a 0
    - c) the set of strings with every 1 followed by two 0s
    - d) the set of strings ending in 00 and not containing 11
    - e) the set of strings containing an even number of 1s
  7. Express each of these sets using a regular expression.
    - a) the set of strings of one or more 0s followed by a 1
    - b) the set of strings of two or more symbols followed by three or more 0s
    - c) the set of strings with either no 1 preceding a 0 or no 0 preceding a 1
    - d) the set of strings containing a string of 1s such that the number of 1s equals 2 modulo 3, followed by an even number of 0s
  8. Construct deterministic finite-state automata that recognize each of these sets from  $I^*$ , where  $I$  is an alphabet.
    - a)  $\emptyset$
    - b)  $\{\lambda\}$
    - c)  $\{a\}$ , where  $a \in I$
  9. Construct nondeterministic finite-state automata that recognize each of the sets in Exercise 8.
  10. Construct nondeterministic finite-state automata that recognize each of these sets.
    - a)  $\{\lambda, 0\}$
    - b)  $\{0, 11\}$
    - c)  $\{0, 11, 000\}$
  - \*11. Show that if  $A$  is a regular set, then  $A^R$ , the set of all reversals of strings in  $A$ , is also regular.
  12. Using the constructions described in the proof of Kleene's theorem, find nondeterministic finite-state automata that recognize each of these sets.
    - a)  $01^*$
    - b)  $(0 \cup 1)1^*$
    - c)  $00(1^* \cup 10)$
  13. Using the constructions described in the proof of Kleene's theorem, find nondeterministic finite-state automata that recognize each of these sets.
    - a)  $0^*1^*$
    - b)  $(0 \cup 11)^*$
    - c)  $01^* \cup 00^*$
  14. Construct a nondeterministic finite-state automaton that recognizes the language generated by the regular grammar  $G = (V, T, S, P)$ , where  $V = \{0, 1, S, A, B\}$ ,  $T = \{0, 1\}$ ,  $S$  is the start symbol, and the set of productions is
    - a)  $S \rightarrow 0A, S \rightarrow 1B, A \rightarrow 0, B \rightarrow 0$ .
    - b)  $S \rightarrow 1A, S \rightarrow 0, S \rightarrow \lambda, A \rightarrow 0B, B \rightarrow 1B, B \rightarrow 1$ .
    - c)  $S \rightarrow 1B, S \rightarrow 0, A \rightarrow 1A, A \rightarrow 0B, A \rightarrow 1, A \rightarrow 0, B \rightarrow 1$ .
- In Exercises 15–17 construct a regular grammar  $G = (V, T, S, P)$  that generates the language recognized by the given finite-state machine.
15. 
  16. 
  17. 
  18. Show that the finite-state automaton constructed from a regular grammar in the proof of Theorem 2 recognizes the set generated by this grammar.
  19. Show that the regular grammar constructed from a finite-state automaton in the proof of Theorem 2 generates the set recognized by this automaton.

20. Show that every nondeterministic finite-state automaton is equivalent to another such automaton that has the property that its starting state is never revisited.
- \*21. Let  $M = (S, I, f, s_0, F)$  be a deterministic finite-state automaton. Show that the language recognized by  $M$ ,  $L(M)$ , is infinite if and only if there is a word  $x$  recognized by  $M$  with  $l(x) \geq |S|$ .
- \*22. One important technique used to prove that certain sets are not regular is the **pumping lemma**. The pumping lemma states that if  $M = (S, I, f, s_0, F)$  is a deterministic finite-state automaton and if  $x$  is a string in  $L(M)$ , the language recognized by  $M$ , with  $l(x) \geq |S|$ , then there are strings  $u$ ,  $v$ , and  $w$  in  $I^*$  such that  $x = uvw$ ,  $l(uv) \leq |S|$  and  $l(v) \geq 1$ , and  $uv^iw \in L(M)$  for  $i = 0, 1, 2, \dots$ . Prove the pumping lemma. [Hint: Use the same idea as was used in Example 5.]
- \*23. Show that the set  $\{0^{2n}1^n \mid n = 0, 1, 2, \dots\}$  is not regular using the pumping lemma given in Exercise 22.
- \*24. Show that the set  $\{1^{n^2} \mid n = 0, 1, 2, \dots\}$  is not regular using the pumping lemma from Exercise 22.
- \*25. Show that the set of palindromes over  $\{0, 1\}$  is not regular using the pumping lemma given in Exercise 22. [Hint: Consider strings of the form  $0^N 1 0^N$ .]
- \*\*26. Show that a set recognized by a finite-state automaton is regular. (This is the “if” part of Kleene’s theorem.)

Suppose that  $L$  is a subset of  $I^*$ , where  $I$  is a nonempty set of symbols. If  $x \in I^*$ , we let  $L/x = \{z \in I^* \mid xz \in L\}$ . We say

that the strings  $x \in I^*$  and  $y \in I^*$  are **distinguishable with respect to  $L$**  if  $L/x \neq L/y$ . A string  $z$  for which  $xz \in L$  but  $yz \notin L$ , or  $xz \notin L$ , but  $yz \in L$  is said to **distinguish**  $x$  and  $y$  with respect to  $L$ . When  $L/x = L/y$ , we say that  $x$  and  $y$  are **indistinguishable** with respect to  $L$ .

27. Let  $L$  be the set of all bit strings that end with 01. Show that 11 and 10 are distinguishable with respect to  $L$  and that the strings 1 and 11 are indistinguishable with respect to  $L$ .
28. Suppose that  $M = (S, I, f, s_0, F)$  is a deterministic finite-state machine. Show that if  $x$  and  $y$  are two strings in  $I^*$  that are distinguishable with respect to  $L(M)$ , then  $f(s_0, x) \neq f(s_0, y)$ .
- \*29. Suppose that  $L$  is a subset of  $I^*$  and for some positive integer  $n$  there are  $n$  strings in  $I^*$  such that every two of these strings are distinguishable with respect to  $L$ . Prove that every deterministic finite-state automaton recognizing  $L$  has at least  $n$  states.
- \*30. Let  $L_n$  be the set of strings with at least  $n$  bits in which the  $n$ th symbol from the end is a 0. Use Exercise 29 to show that a deterministic finite-state machine recognizing  $L_n$  must have at least  $2^n$  states.
- \*31. Use Exercise 29 to show that the language consisting of all bit strings that are palindromes (that is, strings that equal their own reversals) is not regular.

## 13.5 Turing Machines

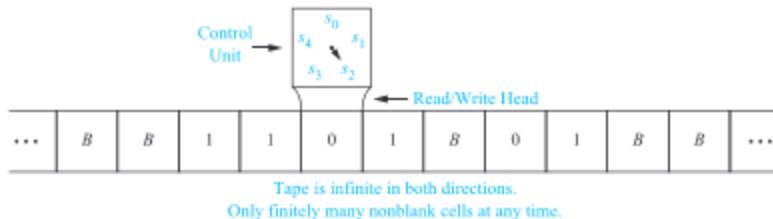
### Introduction



The finite-state automata studied earlier in this chapter cannot be used as general models of computation. They are limited in what they can do. For example, finite-state automata are able to recognize regular sets, but are not able to recognize many easy-to-describe sets, including  $\{0^n 1^n \mid n \geq 0\}$ , which computers recognize using memory. We can use finite-state automata to compute relatively simple functions such as the sum of two numbers, but we cannot use them to compute functions that computers can, such as the product of two numbers. To overcome these deficiencies we can use a more powerful type of machine known as a Turing machine, after Alan Turing, the famous mathematician and computer scientist who invented them in the 1930s.

Basically, a Turing machine consists of a control unit, which at any step is in one of finitely many different states, together with a tape divided into cells, which is infinite in both directions. Turing machines have read and write capabilities on the tape as the control unit moves back and forth along this tape, changing states depending on the tape symbol read. Turing machines are more powerful than finite-state machines because they include memory capabilities that finite-state machines lack. We will show how to use Turing machines to recognize sets, including sets that cannot be recognized by finite-state machines. We will also show how to compute functions using Turing machines. Turing machines are the most general models of computation; essentially, they can do whatever a computer can do. Note that Turing machines are much more powerful than real computers, which have finite memory capabilities.

“Machines take me by surprise with great frequency” – Alan Turing



**FIGURE 1** A Representation of a Turing Machine.

## Definition of Turing Machines

We now give the formal definition of a Turing machine. Afterward we will explain how this formal definition can be interpreted in terms of a control head that can read and write symbols on a tape and move either right or left along the tape.

### DEFINITION 1

A *Turing machine*  $T = (S, I, f, s_0)$  consists of a finite set  $S$  of states, an alphabet  $I$  containing the blank symbol  $B$ , a partial function  $f$  from  $S \times I$  to  $S \times I \times \{R, L\}$ , and a starting state  $s_0$ .

Recall from Section 2.3 that a partial function is defined only for those elements in its domain of definition. This means that for some (state, symbol) pairs the partial function  $f$  may be undefined, but for a pair for which it is defined, there is a unique (state, symbol, direction) triple associated to this pair. We call the five-tuples corresponding to the partial function in the definition of a Turing machine the **transition rules** of the machine.

To interpret this definition in terms of a machine, consider a control unit and a tape divided into cells, infinite in both directions, having only a finite number of nonblank symbols on it at any given time, as pictured in Figure 1. The action of the Turing machine at each step of its operation depends on the value of the partial function  $f$  for the current state and tape symbol.

At each step, the control unit reads the current tape symbol  $x$ . If the control unit is in state  $s$  and if the partial function  $f$  is defined for the pair  $(s, x)$  with  $f(s, x) = (s', x', d)$ , the control unit

1. enters the state  $s'$ ,
2. writes the symbol  $x'$  in the current cell, erasing  $x$ , and
3. moves right one cell if  $d = R$  or moves left one cell if  $d = L$ .

We write this step as the five-tuple  $(s, x, s', x', d)$ . If the partial function  $f$  is undefined for the pair  $(s, x)$ , then the Turing machine  $T$  will *halt*.

A common way to define a Turing machine is to specify a set of five-tuples of the form  $(s, x, s', x', d)$ . The set of states and input alphabet is implicitly defined when such a definition is used.

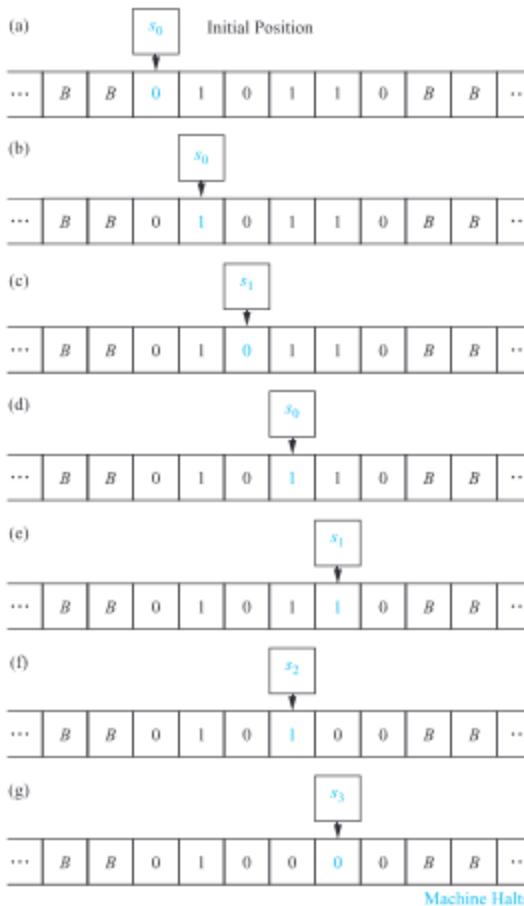
At the beginning of its operation a Turing machine is assumed to be in the initial state  $s_0$  and to be positioned over the leftmost nonblank symbol on the tape. If the tape is all blank, the control head can be positioned over any cell. We will call the positioning of the control head over the leftmost nonblank tape symbol the *initial position* of the machine.

Example 1 illustrates how a Turing machine works.

### EXAMPLE 1



What is the final tape when the Turing machine  $T$  defined by the seven five-tuples  $(s_0, 0, s_0, 0, R)$ ,  $(s_0, 1, s_1, 1, R)$ ,  $(s_0, B, s_3, B, R)$ ,  $(s_1, 0, s_0, 0, R)$ ,  $(s_1, 1, s_2, 0, L)$ ,  $(s_1, B, s_3, B, R)$ , and  $(s_2, 1, s_3, 0, R)$  is run on the tape shown in Figure 2(a)?

FIGURE 2 The Steps Produced by Running  $T$  on the Tape in Figure 1.

**Solution:** We start the operation with  $T$  in state  $s_0$  and with  $T$  positioned over the leftmost nonblank symbol on the tape. The first step, using the five-tuple  $(s_0, 0, s_0, 0, R)$ , reads the 0 in the leftmost nonblank cell, stays in state  $s_0$ , writes a 0 in this cell, and moves one cell right. The second step, using the five-tuple  $(s_0, 1, s_1, 1, R)$ , reads the 1 in the current cell, enters state  $s_1$ , writes a 1 in this cell, and moves one cell right. The third step, using the five-tuple  $(s_1, 0, s_0, 0, R)$ , reads the 0 in the current cell, enters state  $s_0$ , writes a 0 in this cell, and moves one cell right. The fourth step, using the five-tuple  $(s_0, 1, s_1, 1, R)$ , reads the 1 in the current cell, enters state  $s_1$ , writes a 1 in this cell, and moves right one cell. The fifth step, using the five-tuple  $(s_1, 1, s_2, 0, L)$ , reads the 1 in the current cell, enters state  $s_2$ , writes a 0 in this cell, and moves left one cell. The sixth step, using the five-tuple  $(s_2, 1, s_3, 0, R)$ , reads the 1 in the current cell, enters state  $s_3$ , writes a 0 in this cell, and moves right one cell. Finally, in the seventh step, the machine halts because there is no five-tuple beginning with the pair  $(s_3, 0)$  in the description of the machine. The steps are shown in Figure 2.

Note that  $T$  changes the first pair of consecutive 1s on the tape to 0s and then halts. 

## Using Turing Machines to Recognize Sets

Turing machines can be used to recognize sets. To do so requires that we define the concept of a final state as follows. A *final state* of a Turing machine  $T$  is a state that is not the first state in any five-tuple in the description of  $T$  using five-tuples (for example, state  $s_3$  in Example 1).

We can now define what it means for a Turing machine to recognize a string. Given a string, we write consecutive symbols in this string in consecutive cells.

### DEFINITION 2

Let  $V$  be a subset of an alphabet  $I$ . A Turing machine  $T = (S, I, f, s_0)$  *recognizes* a string  $x$  in  $V^*$  if and only if  $T$ , starting in the initial position when  $x$  is written on the tape, halts in a final state.  $T$  is said to recognize a subset  $A$  of  $V^*$  if  $x$  is recognized by  $T$  if and only if  $x$  belongs to  $A$ .

Note that to recognize a subset  $A$  of  $V^*$  we can use symbols not in  $V$ . This means that the input alphabet  $I$  may include symbols not in  $V$ . These extra symbols are often used as markers (see Example 3).

When does a Turing machine  $T$  *not* recognize a string  $x$  in  $V^*$ ? The answer is that  $x$  is not recognized if  $T$  does not halt or halts in a state that is not final when it operates on a tape containing the symbols of  $x$  in consecutive cells, starting in the initial position. (The reader should understand that this is one of many possible ways to define how to recognize sets using Turing machines.)

We illustrate this concept with Example 2.

**EXAMPLE 2** Find a Turing machine that recognizes the set of bit strings that have a 1 as their second bit, that is, the regular set  $(0 \cup 1)1(0 \cup 1)^*$ .

**Solution:** We want a Turing machine that, starting at the leftmost nonblank tape cell, moves right, and determines whether the second symbol is a 1. If the second symbol is 1, the machine should move into a final state. If the second symbol is not a 1, the machine should not halt or it should halt in a nonfinal state.

To construct such a machine, we include the five-tuples  $(s_0, 0, s_1, 0, R)$  and  $(s_0, 1, s_1, 1, R)$  to read in the first symbol and put the Turing machine in state  $s_1$ . Next, we include the five-tuples  $(s_1, 0, s_2, 0, R)$  and  $(s_1, 1, s_3, 1, R)$  to read in the second symbol and either move to state  $s_2$  if this symbol is a 0, or to state  $s_3$  if this symbol is a 1. We do not want to recognize strings that have a 0 as their second bit, so  $s_2$  should not be a final state. We want  $s_3$  to be a final state. So, we can include the five-tuple  $(s_2, 0, s_2, 0, R)$ . Because we do not want to recognize the empty string or a string with one bit, we also include the five-tuples  $(s_0, B, s_2, 0, R)$  and  $(s_1, B, s_2, 0, R)$ .

The Turing machine  $T$  consisting of the seven five-tuples listed here will terminate in the final state  $s_3$  if and only if the bit string has at least two bits and the second bit of the input string is a 1. If the bit string contains fewer than two bits or if the second bit is not a 1, the machine will terminate in the nonfinal state  $s_2$ . 

Given a regular set, a Turing machine that always moves to the right can be built to recognize this set (as in Example 2). To build the Turing machine, first find a finite-state automaton that recognizes the set and then construct a Turing machine using the transition function of the finite-state machine, always moving to the right.

We will now show how to build a Turing machine that recognizes a nonregular set.

**EXAMPLE 3** Find a Turing machine that recognizes the set  $\{0^n 1^n \mid n \geq 1\}$ .

**Solution:** To build such a machine, we will use an auxiliary tape symbol  $M$  as a marker. We have  $V = \{0, 1\}$  and  $I = \{0, 1, M\}$ . We wish to recognize only a subset of strings in  $V^*$ . We will have one final state,  $s_6$ . The Turing machine successively replaces a 0 at the leftmost position of

the string with an  $M$  and a 1 at the rightmost position of the string with an  $M$ , sweeping back and forth, terminating in a final state if and only if the string consists of a block of 0s followed by a block of the same number of 1s.

Although this is easy to describe and is easily carried out by a Turing machine, the machine we need to use is somewhat complicated. We use the marker  $M$  to keep track of the leftmost and rightmost symbols we have already examined. The five-tuples we use are  $(s_0, 0, s_1, M, R)$ ,  $(s_1, 0, s_1, 0, R)$ ,  $(s_1, 1, s_1, 1, R)$ ,  $(s_1, M, s_2, M, L)$ ,  $(s_1, B, s_2, B, L)$ ,  $(s_2, 1, s_3, M, L)$ ,  $(s_3, 1, s_3, 1, L)$ ,  $(s_3, 0, s_4, 0, L)$ ,  $(s_3, M, s_5, M, R)$ ,  $(s_4, 0, s_4, 0, L)$ ,  $(s_4, M, s_0, M, R)$ , and  $(s_5, M, s_6, M, R)$ . For example, the string 000111 would successively become  $M00111$ ,  $M0011M$ ,  $MM011M$ ,  $MM01MM$ ,  $MMM1MM$ ,  $MMMMMM$  as the machine operates until it halts. Only the changes are shown, as most steps leave the string unaltered.

We leave it to the reader (Exercise 13) to explain the actions of this Turing machine and to explain why it recognizes the set  $\{0^n 1^n \mid n \geq 1\}$ .

It can be shown that a set can be recognized by a Turing machine if and only if it can be generated by a type 0 grammar, or in other words, if the set is generated by a phrase-structure grammar. The proof will not be presented here.

## Computing Functions with Turing Machines

A Turing machine can be thought of as a computer that finds the values of a partial function. To see this, suppose that the Turing machine  $T$ , when given the string  $x$  as input, halts with the string  $y$  on its tape. We can then define  $T(x) = y$ . The domain of  $T$  is the set of strings for which  $T$  halts;  $T(x)$  is undefined if  $T$  does not halt when given  $x$  as input. Thinking of a Turing machine as a machine that computes the values of a function on strings is useful, but how can we use Turing machines to compute functions defined on integers, on pairs of integers, on triples of integers, and so on?

To consider a Turing machine as a computer of functions from the set of  $k$ -tuples of non-negative integers to the set of nonnegative integers (such functions are called **number-theoretic functions**), we need a way to represent  $k$ -tuples of integers on a tape. To do so, we use **unary representations** of integers. We represent the nonnegative integer  $n$  by a string of  $n + 1$  1s so that, for instance, 0 is represented by the string 1 and 5 is represented by the string 11111. To represent the  $k$ -tuple  $(n_1, n_2, \dots, n_k)$ , we use a string of  $n_1 + 1$  1s, followed by an asterisk, followed by a string of  $n_2 + 1$  1s, followed by an asterisk, and so on, ending with a string of  $n_k + 1$  1s. For example, to represent the four-tuple  $(2, 0, 1, 3)$  we use the string 111 \* 1 \* 11 \* 1111.

We can now consider a Turing machine  $T$  as computing a sequence of number-theoretic functions  $T, T^2, \dots, T^k, \dots$ . The function  $T^k$  is defined by the action of  $T$  on  $k$ -tuples of integers represented by unary representations of integers separated by asterisks.

**EXAMPLE 4** Construct a Turing machine for adding two nonnegative integers.

**Solution:** We need to build a Turing machine  $T$  that computes the function  $f(n_1, n_2) = n_1 + n_2$ . The pair  $(n_1, n_2)$  is represented by a string of  $n_1 + 1$  1s followed by an asterisk followed by  $n_2 + 1$  1s. The machine  $T$  should take this as input and produce as output a tape with  $n_1 + n_2 + 1$  1s. One way to do this is as follows. The machine starts at the leftmost 1 of the input string, and carries out steps to erase this 1, halting if  $n_1 = 0$  so that there are no more 1s before the asterisk, replaces the asterisk with the leftmost remaining 1, and then halts. We can use these five-tuples to do this:  $(s_0, 1, s_1, B, R)$ ,  $(s_1, *, s_3, B, R)$ ,  $(s_1, 1, s_2, B, R)$ ,  $(s_2, 1, s_2, 1, R)$ , and  $(s_2, *, s_3, 1, R)$ .

Unfortunately, constructing Turing machines to compute relatively simple functions can be extremely demanding. For example, one Turing machine for multiplying two nonnegative integers found in many books has 31 five-tuples and 11 states. If it is challenging to construct Turing machines to compute even relatively simple functions, what hope do we have of building

Turing machines for more complicated functions? One way to simplify this problem is to use a multitape Turing machine that uses more than one tape simultaneously and to build up multitape Turing machines for the composition of functions. It can be shown that for any multitape Turing machine there is a one-tape Turing machine that can do the same thing.

## Different Types of Turing Machines

There are many variations on the definition of a Turing machine. We can expand the capabilities of a Turing machine in a wide variety of ways. For example, we can allow a Turing machine to move right, left, or not at all at each step. We can allow a Turing machine to operate on multiple tapes, using  $(2 + 3n)$ -tuples to describe the Turing machine when  $n$  tapes are used. We can allow the tape to be two-dimensional, where at each step we move up, down, right, or left, not just right or left as we do on a one-dimensional tape. We can allow multiple tape heads that read different cells simultaneously. Furthermore, we can allow a Turing machine to be **nondeterministic**, by allowing a (state, tape symbol) pair to possibly appear as the first elements in more than one five-tuple of the Turing machine. We can also reduce the capabilities of a Turing machine in different ways. For example, we can restrict the tape to be infinite in only one dimension or we can restrict the tape alphabet to have only two symbols. All these variations of Turing machines have been studied in detail.

The crucial point is that no matter which of these variations we use, or even which combination of variations we use, we never increase or decrease the power of the machine. Anything that one of these variations can do can be done by the Turing machine defined in this section, and vice versa. The reason that these variations are useful is that sometimes they make doing some particular job much easier than if the Turing machine defined in Definition 1 were used. They never extend the capability of the machine. Sometimes it is useful to have a wide variety of Turing machines with which to work. For example, one way to show that for every nondeterministic Turing machine, there is a deterministic Turing machine that recognizes the same language is to use a deterministic Turing machine with three tapes. (For details on variations of Turing machines and demonstrations of their equivalence, see [HoMoUl01].)

Besides introducing the notion of a Turing machine, Turing also showed that it is possible to construct a single Turing machine that can simulate the computations of every Turing machine when given an encoding of this target Turing machine and its input. Such a machine is called a **universal Turing machine**. (See a book on the theory of computation, such as [Si06], for more about universal Turing machines.)

## The Church–Turing Thesis



Turing machines are relatively simple. They can have only finitely many states and they can read and write only one symbol at a time on a one-dimensional tape. But it turns out that Turing machines are extremely powerful. We have seen that Turing machines can be built to add numbers and to multiply numbers. Although it may be difficult to actually construct a Turing machine to compute a particular function that can be computed with an algorithm, such a Turing machine can always be found. This was the original goal of Turing when he invented his machines. Furthermore, there is a tremendous amount of evidence for the **Church–Turing thesis**, which states that given any problem that can be solved with an effective algorithm, there is a Turing machine that can solve this problem. The reason this is called a *thesis* rather than a theorem is that the concept of solvability by an effective algorithm is informal and imprecise, as opposed to the notion of solvability by a Turing machine, which is formal and precise. Certainly, though, any problem that can be solved using a computer with a program written in any language, perhaps using an unlimited amount of memory, should be considered effectively solvable. (Note that Turing machines have unlimited memory, unlike computers in the real world, which have only a finite amount of memory.)

Many different formal theories have been developed to capture the notion of effective computability. These include Turing's theory and Church's lambda-calculus, as well as theories proposed by Stephen Kleene and by E. L. Post. These theories seem quite different on the surface. The surprising thing is that they can be shown to be equivalent by demonstrating that they define exactly the same class of functions. With this evidence, it seems that Turing's original ideas, formulated before the invention of modern computers, describe the ultimate capabilities of these machines. The interested reader should consult books on the theory of computation, such as [HoMoU01] and [Si96], for a discussion of these different theories and their equivalence.

For the remainder of this section we will briefly explore some of the consequences of the Church–Turing thesis and we will describe the importance of Turing machines in the study of the complexity of algorithms. Our goal will be to introduce some important ideas from theoretical computer science to entice the interested student to further study. We will cover a lot of ground quickly without providing explicit details. Our discussion will also tie together some of the concepts discussed in previous parts of the book with the theory of computation.

## **Computational Complexity, Computability, and Decidability**

Throughout this book we have discussed the computational complexity of a wide variety of problems. We described the complexity of these problems in terms of the number of operations used by the most efficient algorithms that solve them. The basic operations used by algorithms differ considerably; we have measured the complexity of different algorithms in terms of bit operations, comparisons of integers, arithmetic operations, and so on. In Section 3.3, we defined various classes of problems in terms of their computational complexity. However, these definitions were not precise, because the types of operations used to measure their complexity vary so drastically. Turing machines provide a way to make the concept of computational complexity precise. If the Church–Turing thesis is true, it would then follow that if a problem can be solved using an effective algorithm, then there is a Turing machine that can solve this problem. When a Turing machine is used to solve a problem, the input to the problem is encoded as a string of symbols that is written on the tape of this Turing machine. How we encode input depends on the domain of this input. For example, as we have seen, we can encode a positive integer using a string of 1s. We can also devise ways to express pairs of integers, negative integers, and so on. Similarly, for graph algorithms, we need a way to encode graphs as strings of symbols. This can be done in many ways and can be based on adjacency lists or adjacency matrices. (We omit the details of how this is done.) However, the way input is encoded does not matter as long as it is relatively efficient, as a Turing machine can always change one encoding into another encoding. We will now use this model to make precise some of the notions concerning computational complexity that were informally introduced in Section 3.3.

The kind of problems that are most easily studied by using Turing machines are those problems that can be answered either by a “yes” or by a “no.”

### **DEFINITION 3**

A *decision problem* asks whether statements from a particular class of statements are true. Decision problems are also known as *yes-or-no problems*.

Given a decision problem, we would like to know whether there is an algorithm that can determine whether statements from the class of statements it addresses are true. For example, consider the class of statements each of which asks whether a particular integer  $n$  is prime. This is a decision problem because the answer to the question “Is  $n$  prime?” is either yes or no. Given this decision problem, we can ask whether there is an algorithm that can decide whether each of the statements in the decision problem is true, that is, given an integer  $n$ , deciding whether  $n$  is prime. The answer is that there is such an algorithm. In particular, in Section 3.5 we discussed the algorithm that determines whether a positive integer  $n$  is prime by checking whether it is divisible by primes not exceeding its square root. (There are many other algorithms for determining whether a positive integer is prime.) The set of inputs for which the answer to

the yes–no problem is “yes” is a subset of the set of possible inputs, that is, it is a subset of the set of strings of the input alphabet. In other words, solving a yes–no problem is the same as recognizing the language consisting of all bit strings that represent input values to the problem leading to the answer “yes.” Consequently, solving a yes–no problem is the same as recognizing the language corresponding to the input values for which the answer to the problem is “yes.”

**DECIDABILITY** When there is an effective algorithm that decides whether instances of a decision problem are true, we say that this problem is **solvable** or **decidable**. For instance, the problem of determining whether a positive integer is prime is a solvable problem. However, if no effective algorithm exists for solving a problem, then we say the problem is **unsolvable** or **undecidable**. To show that a decision problem is solvable we need only construct an algorithm that can determine whether statements of the particular class are true. On the other hand, to show that a decision problem is unsolvable we need to prove that no such algorithm exists. (The fact that we tried to find such an algorithm but failed, does not prove the problem is unsolvable.)

By studying only decision problems, it may seem that we are studying only a small set of problems of interest. However, most problems can be recast as decision problems. Recasting the types of problems we have studied in this book as decision problems can be quite complicated, so we will not go into the details of this process here. The interested reader can consult references on the theory of computation, such as [Wo87], which, for example, explains how to recast the traveling salesperson problem (described in Section 9.6) as a decision problem. (To recast the traveling salesman problem as a decision problem, we first consider the decision problem that asks whether there is a Hamilton circuit of weight not exceeding  $k$ , where  $k$  is a positive integer. With some additional effort it is possible to use answers to this question for different values of  $k$  to find the smallest possible weight of a Hamilton circuit.)

In Section 3.1 we introduced the halting problem and proved that it is an unsolvable problem. That discussion was somewhat informal because the notion of a procedure was not precisely defined. A precise definition of the halting problem can be made in terms of Turing machines.

#### DEFINITION 4

The *halting problem* is the decision problem that asks whether a Turing machine  $T$  eventually halts when given an input string  $x$ .

With this definition of the halting problem, we have Theorem 1.

#### THEOREM 1

The halting problem is an unsolvable decision problem. That is, no Turing machine exists that, when given an encoding of a Turing machine  $T$  and its input string  $x$  as input, can determine whether  $T$  eventually halts when started with  $x$  written on its tape.

The proof of Theorem 1 given in Section 3.1 for the informal definition of the halting problem still applies here.

Other examples of unsolvable problems include:

- (i) the problem of determining whether two context-free grammars generate the same set of strings;
- (ii) the problem of determining whether a given set of tiles can be used with repetition allowed to cover the entire plane without overlap; and
- (iii) *Hilbert’s Tenth Problem*, which asks whether there are integer solutions to a given polynomial equation with integer coefficients. (This question occurs tenth on the famous list of 23 problems Hilbert posed in 1900. Hilbert envisioned that the work done to solve these problems would help further the progress of mathematics in the twentieth century. The unsolvability of Hilbert’s Tenth Problem was established in 1970 by Yuri Matiyasevich.)



**COMPUTABILITY** A function that can be computed by a Turing machine is called **computable** and a function that cannot be computed by a Turing machine is called **uncomputable**. It is fairly straightforward, using a countability argument, to show that there are number-theoretic functions that are not computable (see Exercise 39 in Section 2.5). However, it is not so easy to actually produce such a function. The **busy beaver function** defined in the preamble to Exercise 31 is an example of an uncomputable function. One way to show that the busy beaver function is not computable is to show that it grows faster than any computable function. (See Exercise 32.)

Note that every decision problem can be reformulated as the problem of computing a function, namely, the function that has the value 1 when the answer to the problem is “yes” and that has the value 0 when the answer to the problem is “no.” A decision problem is solvable if and only if the corresponding function constructed in this way is computable.

**THE CLASSES P AND NP** In Section 3.3 we informally defined the classes of problems called P and NP. We are now able to define these concepts precisely using the notions of deterministic and nondeterministic Turing machines.

We first elaborate on the difference between a deterministic Turing machine and a nondeterministic Turing machine. The Turing machines we have studied in this section have all been deterministic. In a deterministic Turing machine  $T = (S, I, f, s_0)$ , transition rules are defined by the partial function  $f$  from  $S \times I$  to  $S \times I \times \{R, L\}$ . Consequently, when transition rules of the machine are represented as five-tuples of the form  $(s, x, s', x', d)$ , where  $s$  is the current state,  $x$  is the current tape symbol,  $s'$  is the next state,  $x'$  is the symbol that replaces  $x$  on the tape, and  $d$  is the direction the machine moves on the tape, no two transition rules begin with the same pair  $(s, x)$ .

In a nondeterministic Turing machine, allowed steps are defined using a relation consisting of five-tuples rather than using a partial function. The restriction that no two transition rules begin with the same pair  $(s, x)$  is eliminated; that is, there may be more than one transition rule beginning with each (state, tape symbol) pair. Consequently, in a nondeterministic Turing machine, there is a choice of transitions for some pairs of the current state and the tape symbol being read. At each step of the operation of a nondeterministic Turing machine, the machine picks one of the different choices of the transition rules that begin with the current state and tape symbol pair. This choice can be considered to be a “guess” of which step to use. Just as for deterministic Turing machines, a nondeterministic Turing machine halts when there is no transition rule in its definition that begins with the current state and tape symbol. Given a nondeterministic Turing machine  $T$ , we say that a string  $x$  is recognized by  $T$  if and only if there exists some sequence of transitions of  $T$  that ends in a final state when the machine starts in the initial position with  $x$  written on the tape. The nondeterministic Turing machine  $T$  recognizes the set  $A$  if  $x$  is recognized by  $T$  if and only if  $x \in A$ . The nondeterministic Turing machine  $T$  is said to solve a decision problem if it recognizes the set consisting of all input values for which the answer to the decision problem is yes.

## DEFINITION 5

A decision problem is in P, the *class of polynomial-time problems*, if it can be solved by a deterministic Turing machine in polynomial time in terms of the size of its input. That is, a decision problem is in P if there is a deterministic Turing machine  $T$  that solves the decision problem and a polynomial  $p(n)$  such that for all integers  $n$ ,  $T$  halts in a final state after no more than  $p(n)$  transitions whenever the input to  $T$  is a string of length  $n$ . A decision problem is in NP, the *class of nondeterministic polynomial-time problems*, if it can be solved by a nondeterministic Turing machine in polynomial time in terms of the size of its input. That is, a decision problem is in NP if there is a nondeterministic Turing machine  $T$  that solves the problem and a polynomial  $p(n)$  such that for all integers  $n$ ,  $T$  halts for every choice of transitions after no more than  $p(n)$  transitions whenever the input to  $T$  is a string of length  $n$ .

Problems in P are called **tractable**, whereas problems not in P are called **intractable**. For a problem to be in P, a deterministic Turing machine must exist that can decide in polynomial time whether a particular statement of the class addressed by the decision problem is true. For example, determining whether an item is in a list of  $n$  elements is a tractable problem. (We will not provide details on how this fact can be shown; the basic ideas used in the analyses of algorithms earlier in the text can be adapted when Turing machines are employed.) For a problem to be in NP, it is necessary only that there be a nondeterministic Turing machine that, when given a true statement from the set of statements addressed by the problem, can verify its truth in polynomial time by making the correct guess at each step from the set of allowable steps corresponding to the current state and tape symbol. The problem of determining whether a given graph has a Hamilton circuit is an NP problem, because a nondeterministic Turing machine can easily verify that a simple circuit in a graph passes through each vertex exactly once. It can do this by making a series of correct guesses corresponding to successively adding edges to form the circuit. Because every deterministic Turing machine can also be considered to be a nondeterministic Turing machine where each (state, tape symbol) pair occurs in exactly one transition rule defining the machine, every problem in P is also in NP. In symbols,  $P \subseteq NP$ .

One of the most perplexing open questions in theoretical computer science is whether every problem in NP is also in P, that is, whether  $P = NP$ . As mentioned in Section 3.3, there is an important class of problems, the class of NP-complete problems, such that a problem is in this class if it is in the class NP and if it can be shown that if it is also in the class P, then *every* problem in the class NP must also be in the class P. That is, a problem is NP-complete if the existence of a polynomial-time algorithm for solving it implies the existence of a polynomial-time algorithm for every problem in NP. In this book we have discussed several different NP-complete problems, such as determining whether a simple graph has a Hamilton circuit and determining whether a proposition in  $n$ -variables is a tautology.

## Exercises

1. Let  $T$  be the Turing machine defined by the five-tuples:  $(s_0, 0, s_1, 1, R)$ ,  $(s_0, 1, s_1, 0, R)$ ,  $(s_0, B, s_1, 0, R)$ ,  $(s_1, 0, s_2, 1, L)$ ,  $(s_1, 1, s_1, 0, R)$ , and  $(s_1, B, s_2, 0, L)$ . For each of these initial tapes, determine the final tape when  $T$  halts, assuming that  $T$  begins in initial position.

- a) ... B B 0 0 1 1 B B ...
- b) ... B B 1 0 1 B B B ...
- c) ... B B 1 1 B 0 1 B ...
- d) ... B B B B B B B B B ...

2. Let  $T$  be the Turing machine defined by the five-tuples:  $(s_0, 0, s_1, 0, R)$ ,  $(s_0, 1, s_1, 0, L)$ ,  $(s_0, B, s_1, 1, R)$ ,  $(s_1, 0, s_2, 1, R)$ ,  $(s_1, 1, s_1, 1, R)$ ,  $(s_1, B, s_2, 0, R)$ , and  $(s_2, B, s_3, 0, R)$ . For each of these initial tapes, determine the final tape when  $T$  halts, assuming that  $T$  begins in initial position.

- a) ... B B 0 1 0 1 B B ...
- b) ... B B 1 1 1 B B B ...
- c) ... B B 0 0 B 0 0 B ...
- d) ... B B B B B B B B B ...



**ALONZO CHURCH (1903–1995)** Alonzo Church was born in Washington, D.C. He studied at Göttingen under Hilbert and later in Amsterdam. He was a member of the faculty at Princeton University from 1927 until 1967 when he moved to UCLA. Church was one of the founding members of the Association for Symbolic Logic. He made many substantial contributions to the theory of computability, including his solution to the decision problem, his invention of the lambda-calculus, and, of course, his statement of what is now known as the Church–Turing thesis. Among Church's students were Stephen Kleene and Alan Turing. He published articles past his 90th birthday.

3. What does the Turing machine described by the five-tuples  $(s_0, 0, s_0, 0, R)$ ,  $(s_0, 1, s_1, 0, R)$ ,  $(s_0, B, s_2, B, R)$ ,  $(s_1, 0, s_1, 0, R)$ ,  $(s_1, 1, s_0, 1, R)$ , and  $(s_1, B, s_2, B, R)$  do when given
- 11 as input?
  - an arbitrary bit string as input?
4. What does the Turing machine described by the five-tuples  $(s_0, 0, s_0, 1, R)$ ,  $(s_0, 1, s_0, 1, R)$ ,  $(s_0, B, s_1, B, L)$ ,  $(s_1, 1, s_2, 1, R)$ , do when given
- 101 as input?
  - an arbitrary bit string as input?
5. What does the Turing machine described by the five-tuples  $(s_0, 1, s_1, 0, R)$ ,  $(s_1, 1, s_1, 1, R)$ ,  $(s_1, 0, s_2, 0, R)$ ,  $(s_2, 0, s_3, 1, L)$ ,  $(s_2, 1, s_2, 1, R)$ ,  $(s_3, 1, s_3, 1, L)$ ,  $(s_3, 0, s_4, 0, L)$ ,  $(s_4, 1, s_4, 1, L)$ , and  $(s_4, 0, s_0, 1, R)$  do when given
- 11 as input?
  - a bit string consisting entirely of 1s as input?
6. Construct a Turing machine with tape symbols 0, 1, and  $B$  that, when given a bit string as input, adds a 1 to the end of the bit string and does not change any of the other symbols on the tape.
7. Construct a Turing machine with tape symbols 0, 1, and  $B$  that, when given a bit string as input, replaces the first 0 with a 1 and does not change any of the other symbols on the tape.
8. Construct a Turing machine with tape symbols 0, 1, and  $B$  that, given a bit string as input, replaces all 0s on the tape with 1s and does not change any of the 1s on the tape.
9. Construct a Turing machine with tape symbols 0, 1, and  $B$  that, given a bit string as input, replaces all but the leftmost 1 on the tape with 0s and does not change any of the other symbols on the tape.
10. Construct a Turing machine with tape symbols 0, 1, and  $B$  that, given a bit string as input, replaces the first two consecutive 1s on the tape with 0s and does not change any of the other symbols on the tape.
11. Construct a Turing machine that recognizes the set of all bit strings that end with a 0.
12. Construct a Turing machine that recognizes the set of all bit strings that contain at least two 1s.
13. Construct a Turing machine that recognizes the set of all bit strings that contain an even number of 1s.
14. Show at each step the contents of the tape of the Turing machine in Example 3 starting with each of these strings.
- 0011
  - 00011
  - 101100
  - 000111
15. Explain why the Turing machine in Example 3 recognizes a bit string if and only if this string is of the form  $0^n 1^n$  for some positive integer  $n$ .
- \*16. Construct a Turing machine that recognizes the set  $\{0^{2n} 1^n \mid n \geq 0\}$ .
- \*17. Construct a Turing machine that recognizes the set  $\{0^n 1^n 2^n \mid n \geq 0\}$ .
18. Construct a Turing machine that computes the function  $f(n) = n + 2$  for all nonnegative integers  $n$ .
19. Construct a Turing machine that computes the function  $f(n) = n - 3$  if  $n \geq 3$  and  $f(n) = 0$  for  $n = 0, 1, 2$  for all nonnegative integers  $n$ .
20. Construct a Turing machine that computes the function  $f(n) = n \bmod 3$  for every nonnegative integer  $n$ .
21. Construct a Turing machine that computes the function  $f(n) = 3$  if  $n \geq 5$  and  $f(n) = 0$  if  $n = 0, 1, 2, 3$ , or 4.
22. Construct a Turing machine that computes the function  $f(n) = 2n$  for all nonnegative integers  $n$ .
23. Construct a Turing machine that computes the function  $f(n) = 3n$  for all nonnegative integers  $n$ .
24. Construct a Turing machine that computes the function  $f(n_1, n_2) = n_2 + 2$  for all pairs of nonnegative integers  $n_1$  and  $n_2$ .
- \*25. Construct a Turing machine that computes the function  $f(n_1, n_2) = \min(n_1, n_2)$  for all nonnegative integers  $n_1$  and  $n_2$ .
26. Construct a Turing machine that computes the function  $f(n_1, n_2) = n_1 + n_2 + 1$  for all nonnegative integers  $n_1$  and  $n_2$ .
- Suppose that  $T_1$  and  $T_2$  are Turing machines with disjoint sets of states  $S_1$  and  $S_2$  and with transition functions  $f_1$  and  $f_2$ , respectively. We can define the Turing machine  $T_1 T_2$ , the **composite** of  $T_1$  and  $T_2$ , as follows. The set of states of  $T_1 T_2$  is  $S_1 \cup S_2$ .  $T_1 T_2$  begins in the start state of  $T_1$ . It first executes the transitions of  $T_1$  using  $f_1$  up to, but not including, the step at which  $T_1$  would halt. Then, for all moves for which  $T_1$  halts, it executes the same transitions of  $T_2$  except that it moves to the start state of  $T_2$ . From this point on, the moves of  $T_1 T_2$  are the same as the moves of  $T_2$ .
27. By finding the composite of the Turing machines you constructed in Exercises 18 and 22, construct a Turing machine that computes the function  $f(n) = 2n + 2$ .
28. By finding the composite of the Turing machines you constructed in Exercises 18 and 23, construct a Turing machine that computes the function  $f(n) = 3(n + 2) = 3n + 6$ .
29. Which of the following problems is a decision problem?
- What is the smallest prime greater than  $n$ ?
  - Is a graph  $G$  bipartite?
  - Given a set of strings, is there a finite-state automaton that recognizes this set of strings?
  - Given a checkerboard and a particular type of polyomino (see Section 1.8), can this checkerboard be tiled using polyominoes of this type?
30. Which of the following problems is a decision problem?
- Is the sequence  $a_1, a_2, \dots, a_n$  of positive integers in increasing order?

- b) Can the vertices of a simple graph  $G$  be colored using three colors so that no two adjacent vertices are the same color?
- c) What is the vertex of highest degree in a graph  $G$ ?
- d) Given two finite-state machines, do these machines recognize the same language?

 Let  $B(n)$  be the maximum number of 1s that a Turing machine with  $n$  states with the alphabet  $\{1, B\}$  may print on a tape that is initially blank. The problem of determining  $B(n)$  for particular values of  $n$  is known as the **busy beaver problem**. This problem was first studied by Tibor Rado in 1962. Currently it is known that  $B(2) = 4$ ,  $B(3) = 6$ , and  $B(4) = 13$ , but  $B(n)$

is not known for  $n \geq 5$ .  $B(n)$  grows rapidly; it is known that  $B(5) \geq 4098$  and  $B(6) \geq 3.5 \times 10^{18267}$ .

- \*31. Show that  $B(2)$  is at least 4 by finding a Turing machine with two states and alphabet  $\{1, B\}$  that halts with four consecutive 1s on the tape.
- \*\*32. Show that the function  $B(n)$  cannot be computed by any Turing machine. [Hint: Assume that there is a Turing machine that computes  $B(n)$  in binary. Build a Turing machine  $T$  that, starting with a blank tape, writes  $n$  down in binary, computes  $B(n)$  in binary, and converts  $B(n)$  from binary to unary. Show that for sufficiently large  $n$ , the number of states of  $T$  is less than  $B(n)$ , leading to a contradiction.]

## Key Terms and Results

---

### TERMS

- alphabet (or vocabulary):** a set that contains elements used to form strings
- language:** a subset of the set of all strings over an alphabet
- phrase-structure grammar ( $V, T, S, P$ ):** a description of a language containing an alphabet  $V$ , a set of terminal symbols  $T$ , a start symbol  $S$ , and a set of productions  $P$
- the production  $w \rightarrow w_1$ :**  $w$  can be replaced by  $w_1$  whenever it occurs in a string in the language
- $w_1 \Rightarrow w_2$  ( $w_2$  is directly derivable from  $w_1$ ):**  $w_2$  can be obtained from  $w_1$  using a production to replace a string in  $w_1$  with another string
- $w_1 \stackrel{*}{\Rightarrow} w_2$  ( $w_2$  is derivable from  $w_1$ ):**  $w_2$  can be obtained from  $w_1$  using a sequence of productions to replace strings by other strings
- type 0 grammar:** any phrase-structure grammar
- type 1 grammar:** a phrase-structure grammar in which every production is of the form  $w_1 \rightarrow w_2$ , where  $w_1 = IAr$  and  $w_2 = Iwr$ , where  $A \in N$ ,  $I, r, w \in (N \cup T)^*$  and  $w \neq \lambda$ , or  $w_1 = S$  and  $w_2 = \lambda$  as long as  $S$  is not on the right-hand side of another production
- type 2, or context-free, grammar:** a phrase-structure grammar in which every production is of the form  $A \rightarrow w_1$ , where  $A$  is a nonterminal symbol
- type 3, or regular, grammar:** a phrase-structure grammar where every production is of the form  $A \rightarrow aB$ ,  $A \rightarrow a$ , or  $S \rightarrow \lambda$ , where  $A$  and  $B$  are nonterminal symbols,  $S$  is the start symbol, and  $a$  is a terminal symbol
- derivation (or parse) tree:** an ordered rooted tree where the root represents the starting symbol of a type 2 grammar, internal vertices represent nonterminals, leaves represent terminals, and the children of a vertex are the symbols on the right side of a production, in order from left to right, where the symbol represented by the parent is on the left-hand side
- Backus–Naur form:** a description of a context-free grammar in which all productions having the same nonterminal as their left-hand side are combined with the different right-hand sides of these productions, each separated by a bar, with nonterminal symbols enclosed in angular brackets and the symbol  $\rightarrow$  replaced by  $::=$
- finite-state machine ( $S, I, O, f, g, s_0$ ) (or a Mealy machine):** a six-tuple containing a set  $S$  of states, an input alphabet  $I$ , an output alphabet  $O$ , a transition function  $f$  that assigns a next state to every pair of a state and an input, an output function  $g$  that assigns an output to every pair of a state and an input, and a starting state  $s_0$
- $AB$  (concatenation of  $A$  and  $B$ ):** the set of all strings formed by concatenating a string in  $A$  and a string in  $B$  in that order
- $A^*$  (Kleene closure of  $A$ ):** the set of all strings made up by concatenating arbitrarily many strings from  $A$
- deterministic finite-state automaton ( $S, I, f, s_0, F$ ):** a five-tuple containing a set  $S$  of states, an input alphabet  $I$ , a transition function  $f$  that assigns a next state to every pair of a state and an input, a starting state  $s_0$ , and a set of final states  $F$
- nondeterministic finite-state automaton ( $S, I, f, s_0, F$ ):** a five-tuple containing a set  $S$  of states, an input alphabet  $I$ , a transition function  $f$  that assigns a set of possible next states to every pair of a state and an input, a starting state  $s_0$ , and a set of final states  $F$
- language recognized by an automaton:** the set of input strings that take the start state to a final state of the automaton
- regular expression:** an expression defined recursively by specifying that  $\emptyset$ ,  $\lambda$ , and  $x$ , for all  $x$  in the input alphabet, are regular expressions, and that  $(AB)$ ,  $(A \cup B)$ , and  $A^*$  are regular expressions when  $A$  and  $B$  are regular expressions
- regular set:** a set defined by a regular expression (see page 820)
- Turing machine  $T = (S, I, f, s_0)$ :** a four-tuple consisting of a finite set  $S$  of states, an alphabet  $I$  containing the blank symbol  $B$ , a partial function  $f$  from  $S \times I$  to  $S \times I \times \{R, L\}$ , and a starting state  $s_0$
- nondeterministic Turing machine:** a Turing machine that may have more than one transition rule corresponding to each (state, tape symbol) pair

**decision problem:** a problem that asks whether statements from a particular class of statements are true

**solvable problem:** a problem with the property that there is an effective algorithm that can solve all instances of the problem

**unsolvable problem:** a problem with the property that no effective algorithm exists that can solve all instances of the problem

**computable function:** a function whose values can be computed using a Turing machine

**uncomputable function:** a function whose values cannot be computed using a Turing machine

**P, the class of polynomial-time problems:** the class of problems that can be solved by a deterministic Turing machine in polynomial time in terms of the size of the input

**NP, the class of nondeterministic polynomial-time problems:** the class of problems that can be solved by a nondeter-

ministic Turing machine in polynomial time in terms of the size of the input

**NP-complete:** a subset of the class of NP problems with the property that if any one of them is in the class P, then all problems in NP are in the class P

## RESULTS

For every nondeterministic finite-state automaton there is a deterministic finite-state automaton that recognizes the same set.

**Kleene's theorem:** A set is regular if and only if there is a finite-state automaton that recognizes it.

A set is regular if and only if it is generated by a regular grammar.

The halting problem is unsolvable.

## Review Questions

1. a) Define a phrase-structure grammar.  
b) What does it mean for a string to be derivable from a string  $w$  by a phrase-structure grammar  $G$ ?
  2. a) What is the language generated by a phrase-structure grammar  $G$ ?  
b) What is the language generated by the grammar  $G$  with vocabulary  $\{S, 0, 1\}$ , set of terminals  $T = \{0, 1\}$ , starting symbol  $S$ , and productions  $S \rightarrow 000S, S \rightarrow 1?$   
c) Give a phrase-structure grammar that generates the set  $\{01^n \mid n = 0, 1, 2, \dots\}$ .
  3. a) Define a type 1 grammar.  
b) Give an example of a grammar that is not a type 1 grammar.  
c) Define a type 2 grammar.  
d) Give an example of a grammar that is not a type 2 grammar but is a type 1 grammar.  
e) Define a type 3 grammar.  
f) Give an example of a grammar that is not a type 3 grammar but is a type 2 grammar.
  4. a) Define a regular grammar.  
b) Define a regular language.  
c) Show that the set  $\{0^m 1^n \mid m, n = 0, 1, 2, \dots\}$  is a regular language.
  5. a) What is Backus–Naur form?  
b) Give an example of the Backus–Naur form of the grammar for a subset of English of your choice.
  6. a) What is a finite-state machine?  
b) Show how a vending machine that accepts only quarters and dispenses a soft drink after 75 cents has been deposited can be modeled using a finite-state machine.
  7. Find the set of strings recognized by the deterministic finite-state automaton shown here.
- 
- ```

graph LR
    Start((Start)) -- 0 --> s0((s0))
    s0 -- 0 --> s1((s1))
    s1 -- 0 --> s2((s2))
    s2 -- "0,1" --> s3(((s3)))
    s3 -- "0,1" --> s3
  
```
8. Construct a deterministic finite-state automaton that recognizes the set of bit strings that start with 1 and end with 1.
  9. a) What is the Kleene closure of a set of strings?  
b) Find the Kleene closure of the set  $\{11, 0\}$ .
  10. a) Define a finite-state automaton.  
b) What does it mean for a string to be recognized by a finite-state automaton?
  11. a) Define a nondeterministic finite-state automaton.  
b) Show that given a nondeterministic finite-state automaton, there is a deterministic finite-state automaton that recognizes the same language.
  12. a) Define the set of regular expressions over a set  $I$ .  
b) Explain how regular expressions are used to represent regular sets.
  13. State Kleene's theorem.
  14. Show that a set is generated by a regular grammar if and only if it is a regular set.
  15. Give an example of a set not recognized by a finite-state automaton. Show that no finite-state automaton recognizes it.
  16. Define a Turing machine.
  17. Describe how Turing machines are used to recognize sets.
  18. Describe how Turing machines are used to compute number-theoretic functions.
  19. What is an unsolvable decision problem? Give an example of such a problem.

## Supplementary Exercises

- \*1. Find a phrase-structure grammar that generates each of these languages.

- the set of bit strings of the form  $0^{2n}1^{3n}$ , where  $n$  is a nonnegative integer
- the set of bit strings with twice as many 0s as 1s
- the set of bit strings of the form  $w^2$ , where  $w$  is a bit string

- \*2. Find a phrase-structure grammar that generates the set  $\{0^{2^n} \mid n \geq 0\}$ .

For Exercises 3 and 4, let  $G = (V, T, S, P)$  be the context-free grammar with  $V = \{(), S, A, B\}$ ,  $T = \{(), \}$ , starting symbol  $S$ , and productions  $S \rightarrow A, A \rightarrow AB, A \rightarrow B, B \rightarrow (A),$  and  $B \rightarrow ()$ ,  $S \rightarrow \lambda$ .

3. Construct the derivation trees of these strings.

- $(())$
- $((())$
- $(((()))$

- \*4. Show that  $L(G)$  is the set of all balanced strings of parentheses, defined in the preamble to Supplementary Exercise 55 in Chapter 4.

A context-free grammar is **ambiguous** if there is a word in  $L(G)$  with two derivations that produce different derivation trees, considered as ordered, rooted trees.

- Show that the grammar  $G = (V, T, S, P)$  with  $V = \{0, S\}, T = \{0\}$ , starting state  $S$ , and productions  $S \rightarrow 0S, S \rightarrow S0,$  and  $S \rightarrow 0$  is ambiguous by constructing two different derivation trees for  $0^3$ .
- Show that the grammar  $G = (V, T, S, P)$  with  $V = \{0, S\}, T = \{0\}$ , starting state  $S$ , and productions  $S \rightarrow 0S$  and  $S \rightarrow 0$  is unambiguous.

7. Suppose that  $A$  and  $B$  are finite subsets of  $V^*$ , where  $V$  is an alphabet. Is it necessarily true that  $|AB| = |BA|?$

8. Prove or disprove each of these statements for subsets  $A, B,$  and  $C$  of  $V^*$ , where  $V$  is an alphabet.

- $A(B \cup C) = AB \cup AC$
- $A(B \cap C) = AB \cap AC$
- $(AB)C = A(BC)$
- $(A \cup B)^* = A^* \cup B^*$

9. Suppose that  $A$  and  $B$  are subsets of  $V^*$ , where  $V$  is an alphabet. Does it follow that  $A \subseteq B$  if  $A^* \subseteq B^*?$

10. What set of strings with symbols in the set  $\{0, 1, 2\}$  is represented by the regular expression  $(2^*)(0 \cup (12^*))^*$ ?

The **star height**  $h(E)$  of a regular expression over the set  $I$  is defined recursively by

$$\begin{aligned} h(\emptyset) &= 0; \\ h(x) &= 0 \text{ if } x \in I; \\ h(E_1 \cup E_2) &= h((E_1 E_2)) = \max(h(E_1), h(E_2)) \quad \text{if } E_1 \text{ and } E_2 \text{ are regular expressions;} \\ h(E^*) &= h(E) + 1 \text{ if } E \text{ is a regular expression.} \end{aligned}$$

11. Find the star height of each of these regular expressions.

- $0^*1$
- $0^*1^*$
- $(0^*01)^*$
- $((0^*1)^*)^*$

- e)  $(010^*)(1^*01^*)^*((01)^*(10)^*)^*$

- f)  $((((0^*)1)^*0^*)1)^*$

- \*12. For each of these regular expressions find a regular expression that represents the same language with minimum star height.

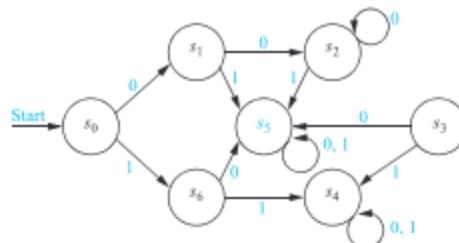
- $(0^*1^*)^*$
- $(0(01^*0))^*$
- $(0^* \cup (01)^* \cup 1^*)^*$

13. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input contains four or more 1s. Then construct a deterministic finite-state automaton that recognizes this set.

14. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input contains four or more consecutive 1s. Then construct a deterministic finite-state automaton that recognizes this set.

15. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input ends with four or more consecutive 1s. Then construct a deterministic finite-state automaton that recognizes this set.

16. A state  $s'$  in a finite-state machine is said to be **reachable** from state  $s$  if there is an input string  $x$  such that  $f(s, x) = s'$ . A state  $s$  is called **transient** if there is no nonempty input string  $x$  with  $f(s, x) = s$ . A state  $s$  is called a **sink** if  $f(s, x) = s$  for all input strings  $x$ . Answer these questions about the finite-state machine with the state diagram illustrated here.



- a) Which states are reachable from  $s_0$ ?

- b) Which states are reachable from  $s_2$ ?

- c) Which states are transient?

- d) Which states are sinks?

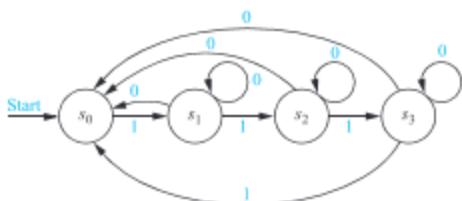
- \*17. Suppose that  $S, I$ , and  $O$  are finite sets such that  $|S| = n, |I| = k,$  and  $|O| = m.$

- How many different finite-state machines (Mealy machines)  $M = (S, I, O, f, g, s_0)$  can be constructed, where the starting state  $s_0$  can be arbitrarily chosen?
- How many different Moore machines  $M = (S, I, O, f, g, s_0)$  can be constructed, where the starting state  $s_0$  can be arbitrarily chosen?

- \*18. Suppose that  $S$  and  $I$  are finite sets such that  $|S| = n$  and  $|I| = k.$  How many different finite-state automata  $M = (S, I, f, s_0, F)$  are there where the starting state  $s_0$

and the subset  $F$  of  $S$  consisting of final states can be chosen arbitrarily

- a) if the automata are deterministic?
  - b) if the automata may be nondeterministic? (Note: This includes deterministic automata.)
19. Construct a deterministic finite-state automaton that is equivalent to the nondeterministic automaton with the state diagram shown here.



20. What is the language recognized by the automaton in Exercise 19?
21. Construct finite-state automata that recognize these sets.
- a)  $0^*(10)^*$
  - b)  $(01 \cup 111)^*10^*(0 \cup 1)$
  - c)  $(001 \cup (11)^*)^*$
- \*22. Find regular expressions that represent the set of all strings of 0s and 1s
- a) made up of blocks of even numbers of 1s interspersed with odd numbers of 0s.
  - b) with at least two consecutive 0s or three consecutive 1s.

- c) with no three consecutive 0s or two consecutive 1s.
- \*23. Show that if  $A$  is a regular set, then so is  $\overline{A}$ .
- \*24. Show that if  $A$  and  $B$  are regular sets, then so is  $A \cap B$ .
- \*25. Find finite-state automata that recognize these sets of strings of 0s and 1s.
- a) the set of all strings that start with no more than three consecutive 0s and contain at least two consecutive 1s
  - b) the set of all strings with an even number of symbols that do not contain the pattern 101
  - c) the set of all strings with at least three blocks of two or more 1s and at least two 0s
- \*26. Show that  $\{0^{2^n} \mid n \in \mathbb{N}\}$  is not regular. You may use the pumping lemma given in Exercise 22 of Section 13.4.
- \*27. Show that  $\{1^p \mid p \text{ is prime}\}$  is not regular. You may use the pumping lemma given in Exercise 22 of Section 13.4.
- \*28. There is a result for context-free languages analogous to the pumping lemma for regular sets. Suppose that  $L(G)$  is the language recognized by a context-free language  $G$ . This result states that there is a constant  $N$  such that if  $z$  is a word in  $L(G)$  with  $l(z) \geq N$ , then  $z$  can be written as  $uvwxy$ , where  $l(vwx) \leq N$ ,  $l(vx) \geq 1$ , and  $uv^iwx^iy$  belongs to  $L(G)$  for  $i = 0, 1, 2, 3, \dots$ . Use this result to show that there is no context-free grammar  $G$  with  $L(G) = \{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$ .
- \*29. Construct a Turing machine that computes the function  $f(n_1, n_2) = \max(n_1, n_2)$ .
- \*30. Construct a Turing machine that computes the function  $f(n_1, n_2) = n_2 - n_1$  if  $n_2 \geq n_1$  and  $f(n_1, n_2) = 0$  if  $n_2 < n_1$ .

## Computer Projects

**Write programs with these input and output.**

1. Given the productions in a phrase-structure grammar, determine which type of grammar this is in the Chomsky classification scheme.
2. Given the productions of a phrase-structure grammar, find all strings that are generated using twenty or fewer applications of its production rules.
3. Given the Backus–Naur form of a type 2 grammar, find all strings that are generated using twenty or fewer applications of the rules defining it.
- \*4. Given the productions of a context-free grammar and a string, produce a derivation tree for this string if it is in the language generated by this grammar.
5. Given the state table of a Moore machine and an input string, produce the output string generated by the machine.
6. Given the state table of a Mealy machine and an input string, produce the output string generated by the machine.
7. Given the state table of a deterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.
8. Given the state table of a nondeterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.
- \*9. Given the state table of a nondeterministic finite-state automaton, construct the state table of a deterministic finite-state automaton that recognizes the same language.
- \*\*10. Given a regular expression, construct a nondeterministic finite-state automaton that recognizes the set that this expression represents.
11. Given a regular grammar, construct a finite-state automaton that recognizes the language generated by this grammar.
12. Given a finite-state automaton, construct a regular grammar that generates the language recognized by this automaton.
- \*13. Given a Turing machine, find the output string produced by a given input string.

## Computations and Explorations

---

Use a computational program or programs you have written to do these exercises.

1. Solve the busy beaver problem for two states by testing all possible Turing machines with two states and alphabet  $\{1, B\}$ .
- \*2. Solve the busy beaver problem for three states by testing all possible Turing machines with three states and alphabet  $\{1, B\}$ .
- \*\*3. Find a busy beaver machine with four states by testing all possible Turing machines with four states and alphabet  $\{1, B\}$ .
- \*\*4. Make as much progress as you can toward finding a busy beaver machine with five states.
- \*\*5. Make as much progress as you can toward finding a busy beaver machine with six states.

## Writing Projects

---

Respond to these with essays using outside sources.

1. Describe how the growth of certain types of plants can be modeled using a Lindenmeyer system. Such a system uses a grammar with productions modeling the different ways plants can grow.
2. Describe the Backus–Naur form (and extended Backus–Naur form) rules used to specify the syntax of a programming language, such as Java, LISP, or Ada, or the database language SQL.
3. Explain how finite-state machines are used by spell-checkers.
4. Explain how finite-state machines are used in the study of network protocols.
5. Explain how finite-state machines are used in speech recognition programs.
6. Compare the use of Moore machines versus Mealy machines in the design of hardware systems and computer software.
7. Explain the concept of minimizing finite-state automata. Give an algorithm that carries out this minimization.
8. Give the definition of cellular automata. Explain their applications. Use the Game of Life as an example.
9. Define a pushdown automaton. Explain how pushdown automata are used to recognize sets. Which sets are recognized by pushdown automata? Provide an outline of a proof justifying your answer.
10. Define a linear-bounded automaton. Explain how linear-bounded automata are used to recognize sets. Which sets are recognized by linear-bounded automata? Provide an outline of a proof justifying your answer.
11. Look up Turing’s original definition of what we now call a Turing machine. What was his motivation for defining these machines?
12. Describe the concept of the universal Turing machine. Explain how such a machine can be built.
13. Explain the kinds of applications in which nondeterministic Turing machines are used instead of deterministic Turing machines.
14. Show that a Turing machine can simulate any action of a nondeterministic Turing machine.
15. Show that a set is recognized by a Turing machine if and only if it is generated by a phrase-structure grammar.
16. Describe the basic concepts of the lambda-calculus and explain how it is used to study computability of functions.
17. Show that a Turing machine as defined in this chapter can do anything a Turing machine with  $n$  tapes can do.
18. Show that a Turing machine with a tape infinite in one direction can do anything a Turing machine with a tape infinite in both directions can do.



## Bibliography

1. Paul R. Halmos, I want to be a mathematician, Springer-Verlag, New York, 1985.
2. Richard Johnsonbaugh, Discrete mathematics, eighth ed., Pearson, 2018.
3. Yiannis Moschovakis, Notes on set theory, second ed., Undergraduate Texts in Mathematics, Springer, New York, 2006.
4. Kenneth H. Rosen, Discrete Mathematics and Its Applications, McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2012.
5. Michiel Smid, Discrete Structures for Computer Science, Counting, Recursion, and Probability, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, 2019.
6. Gary Haggad, John Schlipf, Sue Whitesides, Discrete Mathematics for Computer Science, Bob Pirl, 2004.
7. Glynn Winskel, Discrete Mathematics II: Set Theory for Computer Science, Part IA Comp. Sci. Lecture Notes, February 2012.
8. Rafael Pass, Wei-Lung Dustin Tseng, A Course in Discrete Structures, 2012.