INTRODUCTION TO THE THEORY OF COMPUTATION

You are about to embark on the study of a fascinating and important subject: the theory of computation. It comprises the fundamental mathematical properties of computer hardware, software, and certain applications thereof. In studying this subject, we seek to determine what can and cannot be computed, how quickly, with how much memory, and on which type of computational model. The subject has obvious connections with engineering practice, and, as in many sciences, it also has purely philosophical aspects.

AUTOMATA, COMPUTABILITY, AND COMPLEXITY

This course focuses on three traditionally central areas of the theory of computation: automata, computability, and complexity. They are linked by the question:

What are the fundamental capabilities and limitations of computers?

In each of the three areas—automata, computability, and complexity—this question is interpreted differently, and the answers vary according to the interpretation.
Here, we introduce these parts in reverse order because by starting from the end you can better understand the reason for the beginning.

COMPLEXITY THEORY

Computer problems come in different varieties; some are easy, and some are hard. For example, the sorting problem is an easy one. Say that you need to arrange a list of numbers in ascending order. Even a small computer can sort a million numbers rather quickly. Compare that to a scheduling problem. Say that you must find a schedule of classes for the entire university to satisfy some reasonable constraints, such as that no two classes take place in the same room at the same time. The scheduling problem seems to be much harder than the sorting problem. If you have just a thousand classes, finding the best schedule may require centuries, even with a supercomputer.

What makes some problems computationally hard and others easy?

This is the central question of complexity theory. Remarkably, we don't know the answer to it, though it has been intensively researched for over 40 years. Later, we explore this interesting question and some of its consequences.

In one important achievement of complexity theory thus far, researchers have discovered a well-designed scheme for classifying problems according to their computational difficulty. Using this scheme, we can demonstrate a method for giving evidence that certain problems are computationally hard, even if we are unable to prove that they are.

You have several options when you confront a problem that appears to be computationally hard. First, by understanding which aspect of the problem is at the root of the difficulty, you may be able to alter it so that the problem is more easily solvable. Second, you may be able to settle for less than a perfect solution to the problem. In certain cases, finding solutions that only

approximate the perfect one is relatively easy. Third, some problems are hard only in the worst case situation, but easy most of the time. Depending on the application, you may be satisfied with a procedure that occasionally is slow but usually runs quickly. Finally, you may consider alternative types of computation, such as randomized computation, that can speed up certain tasks.

One applied area that has been affected directly by complexity theory is the ancient field of cryptography. In most fields, an easy computational problem is preferable to a hard one because easy ones are cheaper to solve. Cryptography is unusual because it specifically requires computational problems that are hard, rather than easy. Secret codes should be hard to break without the secret key or password. Complexity theory has pointed cryptographers in the direction of computationally hard problems around which they have designed revolutionary new codes.

COMPUTABILITY THEORY

Certain basic problems cannot be solved by computers. One example of this phenomenon is the problem of determining whether a mathematical statement is true or false. This task is the bread and butter of mathematicians. It seems like a natural for solution by computer because it lies strictly within the area of mathematics. But no computer algorithm can perform this task.

The theories of computability and complexity are closely related. In complexity theory, the objective is to classify problems as easy ones and hard ones; whereas in computability theory, the classification of problems is by those that are solvable and those that are not. Computability theory introduces several of the concepts used in complexity theory.

AUTOMATA THEORY

Automata theory deals with the definitions and properties of mathematical models of computation. These models play a role in several applied areas of computer science. One model, called the **finite automaton**, is used in text processing, compilers, and hardware design. Another model, called the **context-free grammar**, is used in programming languages and artificial intelligence.

Automata theory is an excellent place to begin the study of the theory of computation. The theories of computability and complexity require a precise definition of a **computer**. Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other nontheoretical areas of computer science.

MATHEMATICAL NOTIONS AND TERMINOLOGY

As in any mathematical subject, we begin with a discussion of the basic mathematical objects, tools, and notation that we expect to use.

SETS

A **set** is a group of objects represented as a unit. Sets may contain any type of object, including numbers, symbols, and even other sets. The objects in a set are called its **elements** or **members**. Sets may be described formally in several ways.

One way is by listing a set's elements inside braces. Thus the set

$S = \{7, 21, 57\}$

contains the elements 7, 21, and 57. The symbols $\in$ and $\notin$ denote set membership and nonmembership. We write $7 \in \{7, 21, 57\}$ and $8 \notin \{7, 21, 57\}$. For two sets A and B, we say that A is a **subset** of B, written $A \subseteq B$, if every member of A also is a member of B. We say that A is a **proper subset** of B, written $A \subseteq B$, if A is a subset of B and not equal to B.
The order of describing a set doesn't matter, nor does repetition of its members. We get the same set S by writing {57, 7, 7, 7, 21}.

An **infinite set** contains infinitely many elements. We cannot write a list of all the elements of an infinite set, so we sometimes use the ". . ." notation to mean "continue the sequence forever." Thus we write the set of natural numbers N as

$\{1, 2, 3, \ldots\}$.

The set with zero members is called the **empty set** (**or null set)** and is written $\emptyset$. A set with one member is sometimes called a **singleton set**, and a set with two members is called an **unordered pair**.

When we want to describe a set containing elements according to some rule, we write {n| rule about n}. Thus $\{n \mid n = m^2 \text{ for some } m \in N\}$ means the set of perfect squares.

If we have two sets A and B, the **union** of A and B, written $A \cup B$, is the set we get by combining all the elements in A and B into a single set. The **intersection** of A and B, written $A \cap B$, is the set of elements that are in both A and B. The complement of A, written $\bar{A}$, is the set of all elements under consideration that are not in A.

As is often the case in mathematics, a picture helps clarify a concept. For sets, we use a type of picture called a **Venn diagram**. It represents sets as regions enclosed by circular lines.

SEQUENCES AND TUPLES

A **sequence** of objects is a list of these objects in some order. We usually designate a sequence by writing the list within parentheses. For example, the sequence 7, 21, 57 would be written

(7, 21, 57).

The order doesn't matter in a set, but in a sequence it does. Hence (7, 21, 57) is not the same as (57, 7, 21). Similarly, repetition does matter in a sequence, but it doesn't matter in a set. Thus (7, 7, 21, 57) is different from both of the other sequences, whereas the set {7, 21, 57} is identical to the set {7, 7, 21, 57}.

Sets and sequences may appear as elements of other sets and sequences. For example, the **power set** of A is the set of all subsets of A. If A is the set {0, 1}, the power set of A is the set { ∅, {0}, {1}, {0, 1} }. The set of all ordered pairs whose elements are 0s and 1s is { (0, 0), (0, 1), (1, 0), (1, 1) }.

If A and B are two sets, the **Cartesian product** or **cross product** of A and B, written $A \times B$, is the set of all ordered pairs wherein the first element is a member of A and the second element is a member of B.

EXAMPLE 0.5
If A = {1, 2} and B = {x, y, z},
$A \times B$ = { (1, x), (1, y), (1, z), (2, x), (2, y), (2, z) }.

EXAMPLE 0.6
If A and B are as in Example 0.5,
$A \times B \times A$ = {(1, x, 1), (1, x, 2), (1, y, 1), (1, y, 2), (1, z, 1), (1, z, 2),
(2, x, 1), (2, x, 2), (2, y, 1), (2, y, 2), (2, z, 1), (2, z, 2)}

FUNCTIONS AND RELATIONS

Functions are central to mathematics. A **function** is an object that sets up an input–output relationship. A function takes an input and produces an output. In every function, the same input always produces the same output. If f is a function whose output value is b when the input value is a, we write
f(a) = b.
.
A function also is called a **mapping**, and, if f(a) = b, we say that f maps a to b.
The set of possible inputs to the function is called its **domain**. The outputs of a function come from a set called its **range**. The notation for saying that f is a function with domain D and range R is
$f : D \rightarrow R$.

We may describe a specific function in several ways. One way is with a procedure for computing an output from a specified input. Another way is with a table that lists all possible inputs and gives the output for each input.

EXAMPLE 0.8
Consider the function $f : \{0, 1, 2, 3, 4\} \rightarrow \{0, 1, 2, 3, 4\}$.

n	f(n)
0	1
1	2
2	3
3	4
4	0

This function adds 1 to its input and then outputs the result modulo 5. A number modulo m is the remainder after division by m.

EXAMPLE 0.9
Sometimes a two-dimensional table is used if the domain of the function is the Cartesian product of two sets. Here is another function, g : Z4×Z4→Z4. The entry at the row labeled i and the column labeled j in the table is the value of g(i, j).

g	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

The function g is the addition function modulo 4.

A **predicate** or **property** is a function whose range is {TRUE, FALSE}. For example, let even be a property that is TRUE if its input is an even number and FALSE if its input is an odd number. Thus even(4) = TRUE and even(5) = FALSE.

GRAPHS

An **undirected graph**, or simply a **graph**, is a set of points with lines connecting some of the points. The points are called **nodes** or **vertices**, and the lines are called **edges**, as shown in the following figure.

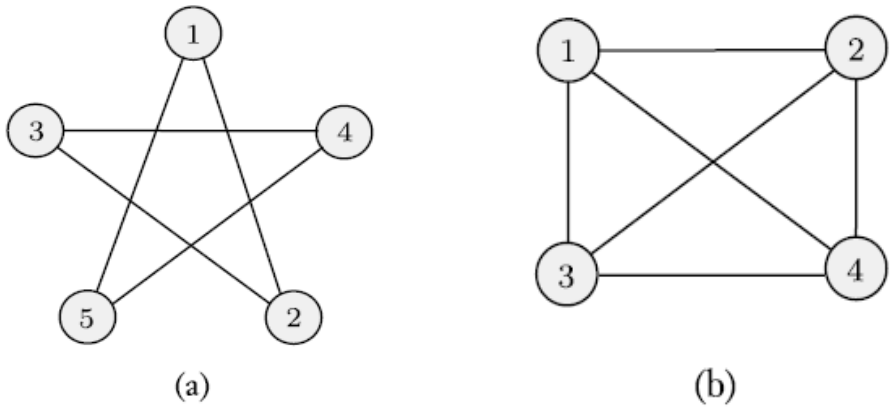

FIGURE **0.12:** Examples of graphs
The number of edges at a particular node is the **degree** of that node. In Figure 0.12(a), all the nodes have degree 2. In Figure 0.12(b), all the nodes have degree 3. No more than one edge is allowed between any two nodes. We may allow an edge from a node to itself, called a **self-loop**, depending on the situation.

In a graph G that contains nodes i and j, the pair (i, j) represents the edge that connects i and j. The order of i and j doesn't matter in an undirected graph, so the pairs (i, j) and (j, i) represent the same edge. Sometimes we describe undirected edges with unordered pairs using set notation as in {i, j}. If V is the set of nodes of G and E is the set of edges, we say G = (V, E). We can describe a graph with a diagram or more formally by specifying V and E. For example, a formal description of the graph in Figure 0.12(a) is

({1, 2, 3, 4, 5}, {(1, 2), (2, 3), (3, 4), (4, 5), (5, 1)}),

and a formal description of the graph in Figure 0.12(b) is

({1, 2, 3, 4}, {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}).

Graphs frequently are used to represent data. Nodes might be cities and edges the connecting highways, or nodes might be people and edges the friendships between them. Sometimes, for convenience, we label the nodes and/or edges of a graph, which then is called a **labeled graph**. Figure 0.13 depicts a graph whose nodes are cities and whose edges are labeled with the dollar cost of the cheapest nonstop airfare for travel between those cities if flying nonstop between them is possible.

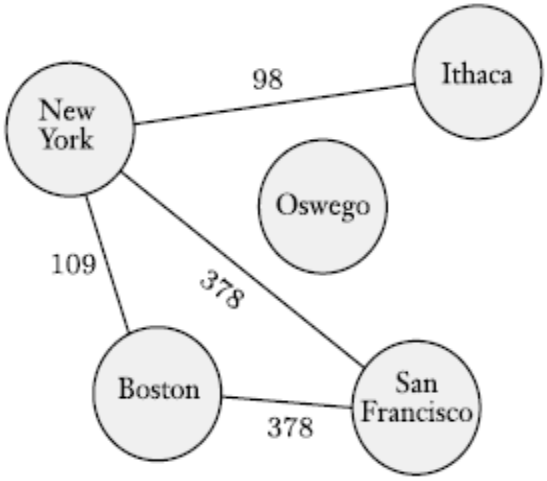

FIGURE 0.13: Cheapest nonstop airfares between various cities

A **path** in a graph is a sequence of nodes connected by edges. A **simple path** is a path that doesn't repeat any nodes. A graph is **connected** if every two nodes have a path between them. A path is a **cycle** if it starts and ends in the same node. A **simple cycle** is one that contains at least three nodes and repeats only the first and last nodes. A graph is a **tree** if it is connected and has no simple cycles, as shown in Figure 0.15. A tree may contain a specially designated node called the **root**. The nodes of degree 1 in a tree, other than the root, are called the **leaves** of the tree.

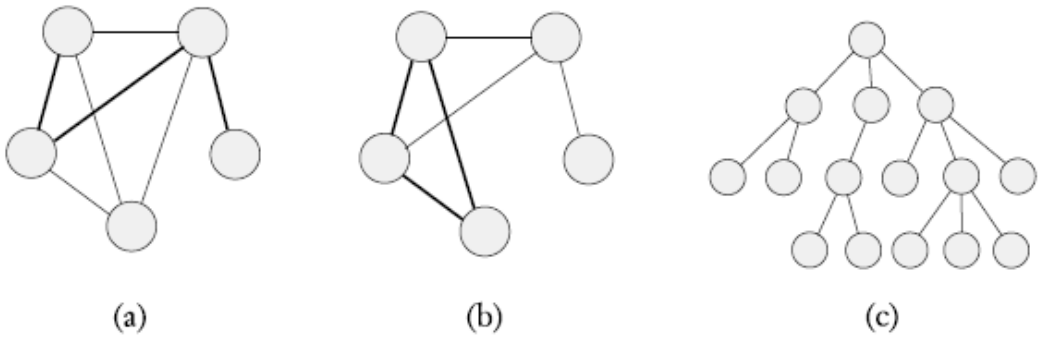

FIGURE **0.15**
(a) A path in a graph, (b) a cycle in a graph, and (c) a tree

A **directed graph** has arrows instead of lines, as shown in the following figure. The number of arrows pointing from a particular node is the **outdegree** of that node, and the number of arrows pointing to a particular node is the **indegree**.

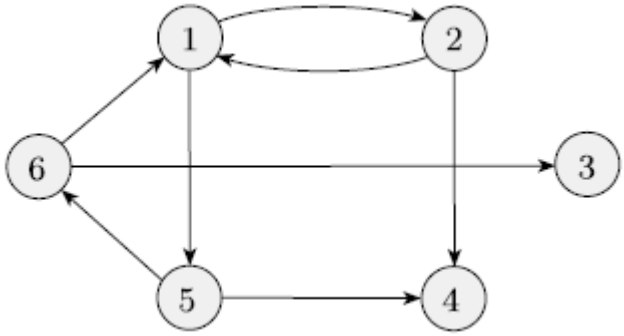

FIGURE 0.16: A directed graph

In a directed graph, we represent an edge from i to j as a pair (i, j). The formal description of a directed graph G is (V,E), where V is the set of nodes and E is the set of edges. The formal description of the graph in Figure 0.16 is

({1,2,3,4,5,6}, {(1,2), (1,5), (2,1), (2,4), (5,4), (5,6), (6,1), (6,3)}).

A path in which all the arrows point in the same direction as its steps is called a **directed path**. A directed graph is **strongly connected** if a directed path connects every two nodes.

STRINGS AND LANGUAGES

Strings of characters are fundamental building blocks in computer science. The alphabet over which the strings are defined may vary with the application. For our purposes, we define an alphabet to be any nonempty finite set. The members of the alphabet are the symbols of the

alphabet. We generally use capital Greek letters $\sum$ and Γ to designate alphabets and a typewriter font for symbols from an alphabet. The following are a few examples of alphabets.

$$\Sigma_1 = \{0,1\}$$

$$\Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

$$\Gamma = \{0, 1, x, y, z\}$$

A **string over an alphabet** is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas. If $\sum_1 = \{0,1\}$,
then 01001 is a string over $\sum_1$. If $\sum_2 = \{a, b, c, \ldots, z\}$, then abracadabra is a string over $\sum_2$. If w is a string over $\sum$, the **length** of w, written |w|, is the number of symbols that it contains. The string of length zero is called the **empty string** and is written ε. The empty string plays the role of 0 in a number system. If w has length n, we can write $w = w_1 w_2 \cdots w_n$ where each $w_i \in \sum$. String z is a substring of w if z appears consecutively within w. For example, cad is a substring of abracadabra.

If we have string x of length m and string y of length n, the concatenation of x and y, written xy, is the string obtained by appending y to the end of x.

The **lexicographic order** of strings is the same as the familiar dictionary order. We'll occasionally use a modified lexicographic order, called shortlex order or simply **string order**, that is identical to lexicographic order, except that shorter strings precede longer strings. Thus the string ordering of all strings over the alphabet {0,1} is

(ε, 0, 1, 00, 01, 10, 11, 000, . . .).

Say that string x is a **prefix** of string y if a string z exists where xz = y, and that
x is a **proper prefix** of y if in addition $x \neq y$. A **language** is a set of strings. A language is **prefix-free** if no member is a proper prefix of another member.

BOOLEAN LOGIC

Boolean logic is a mathematical system built around the two values TRUE and FALSE. Though originally conceived of as pure mathematics, this system is now considered to be the foundation of digital electronics and computer design. The values TRUE and FALSE are called the **Boolean values** and are often represented by the values 1 and 0. We use Boolean values in situations with two possibilities, such as a wire that may have a high or a low voltage, a proposition that may be true or false, or a question that may be answered yes or no.

We can manipulate Boolean values with the **Boolean operations**. The simplest Boolean operation is the **negation** or **NOT** operation, designated with the symbol ¬. The negation of a Boolean value is the opposite value. Thus ¬0 = 1 and ¬1 = 0. We designate the **conjunction** or **AND** operation with the symbol ∧. The conjunction of two Boolean values is 1 if both of those values are 1. The **disjunction** or **OR** operation is designated with the symbol ∨. The disjunction

of two Boolean values is 1 if either of those values is 1. We summarize this information as follows.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \neg 0 = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \neg 1 = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

We use Boolean operations for combining simple statements into more complex Boolean expressions, just as we use the arithmetic operations + and × to construct complex arithmetic expressions. For example, if P is the Boolean value representing the truth of the statement "the sun is shining" and Q represents the truth of the statement "today is Monday", we may write P ∧ Q to represent the truth value of the statement "the sun is shining and today is Monday" and similarly
for P ∨ Q with and replaced by or. The values P and Q are called the operands of the operation.

Several other Boolean operations occasionally appear. The **exclusive or**, or **XOR**, operation is designated by the ⊕ symbol and is 1 if either but not both of its two operands is 1. The **equality** operation, written with the symbol ↔, is 1if both of its operands have the same value. Finally, the **implication** operation is designated by the symbol → and is 0 if its first operand is 1 and its second operand is 0; otherwise, → is 1. We summarize this information as follows.

$$\begin{array}{lll} 0 \oplus 0 = 0 & 0 \leftrightarrow 0 = 1 & 0 \rightarrow 0 = 1 \\ 0 \oplus 1 = 1 & 0 \leftrightarrow 1 = 0 & 0 \rightarrow 1 = 1 \\ 1 \oplus 0 = 1 & 1 \leftrightarrow 0 = 0 & 1 \rightarrow 0 = 0 \\ 1 \oplus 1 = 0 & 1 \leftrightarrow 1 = 1 & 1 \rightarrow 1 = 1 \end{array}$$

We can establish various relationships among these operations. In fact, we can express all Boolean operations in terms of the AND and NOT operations, as the following identities show. The two expressions in each row are equivalent. Each row expresses the operation in the left-hand column in terms of operations above it and AND and NOT.

$$\begin{array}{ll} P \vee Q & \neg(\neg P \wedge \neg Q) \\ P \rightarrow Q & \neg P \vee Q \\ P \leftrightarrow Q & (P \rightarrow Q) \wedge (Q \rightarrow P) \\ P \oplus Q & \neg(P \leftrightarrow Q) \end{array}$$

The **distributive law** for AND and OR comes in handy when we manipulate Boolean expressions. It is similar to the distributive law for addition and multiplication, which states that a ×(b+c) = (a ×b)+(a ×c). The Boolean version
comes in two forms:

- $P \wedge (Q \vee R)$ equals $(P \wedge Q) \vee (P \wedge R)$, and its dual
- $P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$.

CP 214: ASSIGNMENT 1

1. What are the three major areas of the theory of computation?
2. Why is theory of computation important in computer science?
3. Describe three strategies for handling computationally hard problems.
4. Explain the relationship between complexity theory and cryptography.
5. What is the purpose of classifying problems according to difficulty?
6. Why are approximations sometimes acceptable in solving hard problems?
7. How do sequences differ from sets?
8. If A = {1, 2} and B = {x, y}, list A × B.
9. List all elements of:

   A × B × A, where A = {0,1} and B = {x,y}.
10. If A has a elements and B has b elements, how many elements are in A × B? Explain your answer.
11. If C is a set with c elements, how many elements are in the power set of C? Explain your answer.
12. Let X be the set {1, 2, 3, 4, 5} and Y be the set {6, 7, 8, 9, 10}. The unary function f : X→Y and the binary function g : X ×Y →Y are described in the following tables.

n	f(n)
1	6
2	7
3	6
4	7
5	6

g	6	7	8	9	10
1	10	10	10	10	10
2	7	8	9	10	6
3	7	7	8	8	9
4	9	8	7	6	10
5	6	6	6	6	6

   a. What is the value of f(2)?
   b. What are the range and domain of f?
   c. What is the value of g(2, 10)?

d. What are the range and domain of g?
e. What is the value of g(4, f(4))?

13. Describe how a function can be represented using a table.

14. Given graph G = ({1,2,3}, {(1,2), (2,3)}), list:

- Degrees of all nodes
- All paths from 1 to 3

15. Is "cad" a substring of "abracadabra"? Justify.