Zhiwei Xu · Jialin Zhang

Computational Thinking: A Perspective on Computer Science

Springer

Computational Thinking: A Perspective on Computer Science

Zhiwei Xu • Jialin Zhang

Computational Thinking: A Perspective on Computer Science

Springer

Zhiwei Xu
University of Chinese Academy
of Sciences
Beijing, China

Jialin Zhang
University of Chinese Academy
of Sciences
Beijing, China

ISBN 978-981-16-3847-3 ISBN 978-981-16-3848-0 (eBook)
https://doi.org/10.1007/978-981-16-3848-0

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

To Hao, for continuous inspiration and support
Zhiwei

To my husband and little son, with love
Jialin

Preface

This book provides an introduction to computer science from the computational thinking perspective. It explains the way of thinking in computer science through chapters of logic thinking, algorithmic thinking, systems thinking, and network thinking. It is purposely designed as a textbook for the first computer science course serving undergraduates from all disciplines.

The book focuses on elementary knowledge such that all material can be covered in a one-semester course of Introduction to Computer Science. It is designed for all students, assuming no prior programming experience. At the same time, students with prior programming experience should not find the course boring.

The book practices an active learning method, utilizing recent advice by Donald Knuth: “The ultimate test of whether I understand something is if I can explain it to a computer.” The book is designed to enable students to rise from the basement level of *remembering* to the top level of *creating* in Bloom’s taxonomy of education objectives. More than 200 hands-on exercises, thought experiments, and projects are included to encourage students to create. Examples of creative tasks include:

- Design a Turing machine to do n-bit addition, where n could be arbitrarily large. This could be a student’s first design of an abstract computer.
- Design a team computer to do quicksort. This could be a student’s first design of a real computer, including its instruction set and machine organization.
- Develop a computer application (a steganography computer program) to hide a text file hamlet.txt in a picture file Autumn.bmp.
- Design a smart algorithm and a program to compute Fibonacci numbers F(n), where n could be as large as one million or even one billion.
- Create a dynamic webpage of creative expression for a *Kitty Band*, which can play a piece of music given an input string of music scores.

The material of the book has been used in the University of Chinese Academy of Sciences since 2014, serving a required course for freshmen from all schools. It was also used in summer schools organized by China Computer Federation, to train university and high-school instructors on teaching a Computer Fundamentals course utilizing computational thinking.

Supplementary material is provided at cs101.ucas.edu.cn.

Beijing, China
October 2021

Zhiwei Xu
Jialin Zhang

Introduction

This textbook is for a one-semester course of Introduction to Computer Science (e.g., CS101), targeting undergraduate students from all disciplines. It is a self-contained book with no prerequisites. The little prior knowledge and notations needed are explained along the way and summarized in Appendices.

The book is designed to introduce elementary knowledge of computer science and the field's way of thinking. It has the following four objectives and features:

- *Embodying computational thinking*. The way of thinking in computer science is characterized by three features without and eight understandings within. Introductory bodies of knowledge are organized into chapters of logic thinking, algorithmic thinking, systems thinking, and network thinking.
- *Aiming at upper levels of Bloom's taxonomy*, with a significant portion of learning material going from "remember" to "create," as shown in Fig. 1. The learning method uses Knuth's Test: "The ultimate test of whether I understand something is if I can explain it to a computer." More than 200 hands-on exercises and thought experiments are included to encourage students to create.
- *Focusing on elementary knowledge without dumbing down*. An explicit goal is that all material should be coverable in one semester, for a class of hundreds of students of all disciplines, assuming no prior programming experiences. At the same time, experienced students should not find the course boring.
- *Learning from a decade of educational experience*. We spent 4 years designing the course and 6 years teaching the material. The contents have gone through three major revisions. For instance, version 1 has no programming. Version 2 includes Go language programming contents. Version 3 (the current version) requires a student to write roughly 300 lines of Go code and 100 lines of Web code, where most students can learn Web programming by themselves.

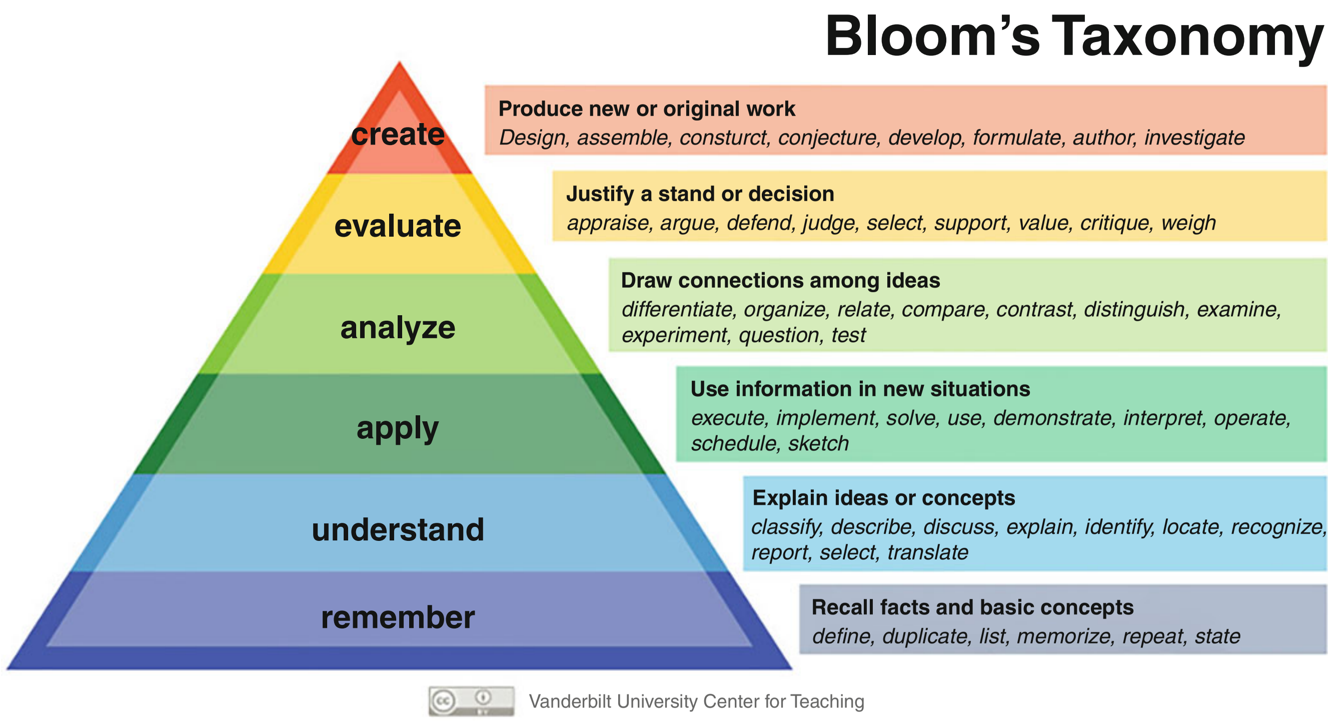

Bloom's Taxonomy
create
evaluate
analyze
apply
understand
remember
Produce new or original work
Design, assemble, consturct, conjecture, develop, formulate, author, investigate
Justify a stand or decision
appraise, argue, defend, judge, select, support, value, critique, weigh
Draw connections among ideas
differentiate, organize, relate, compare, contrast, distinguish, examine, experiment, question, test
Use information in new situations
execute, implement, solve, use, demonstrate, interpret, operate, schedule, sketch
Explain ideas or concepts
classify, describe, discuss, explain, identify, locate, recognize, report, select, translate
Recall facts and basic concepts
define, duplicate, list, memorize, repeat, state
Vanderbilt University Center for Teaching

Fig. 1 Bloom's taxonomy of educational objectives. (Figure showing Bloom's taxonomy by Vandy CFT is licensed under CC BY 2.0)

Problem-Solving Examples

Computer science is a subject studying computational processes in problem-solving and creative expression. This textbook includes over 200 problems as examples, exercises, and hands-on projects. They provide a glimpse of how computers work and what kinds of problem-solving and creative expression are enabled by computer science and computational thinking.

The book shows how to solve such problems. In doing so, it introduces elementary knowledge on not only how to use a computer but also how to design a computer. We demonstrate that computer science is intellectually interesting, by heeding Donald Knuth's advice: "In most of life, you can bluff, but not with computers." We take special care to avoid underestimating the potentials of students and dumbing down the course material.

Six representative problems are shown below and illustrated in Fig. 2.

- Design a Turing machine to do n-bit addition, where n could be arbitrarily large, such as $n = 2^3 = 8$, $n = 2^{10} = 1024$, or $n = 2^{20} = 1048576$. This could be a student's first **abstract computer**.
- Design a human-computer to do quicksort. A student is asked to design a team computer to successfully rearrange a group of students ordered by students' names to another group ordered by students' heights, as shown in Fig. 2a. This could be a student's first design of a working **real compute**r, complete with its instruction set and machine organization of essential components.
- Compute the area of a panda. Computer science offers new abilities to solve problems beyond ordinary school math, such as computing the irregular area of the panda picture in Fig. 2b. The same **computer application** idea extends to irregular shapes of multiple dimensions, and "area" can be replaced by volume, mass, energy, number of particles, etc.
- Compute Fibonacci numbers F(n), where n could be as large as one billion. This problem reveals how smart **algorithms**, together with systems support, can drastically reduce computing time, from 2^n to n or even log$n$.
- Prove a problem belonging to P or NP, intuitively. Students are asked to prove whether a simple problem belongs to P or NP. For such algorithmic **complexity** material to be included in an introductory course, the problem and the proof must be intuitively simple, involving only elementary mathematics and a short reasoning sequence. A sample problem is the following: decide if $2n$ numbers can be divided into two groups, each having n numbers, such that the sums of the two groups are equal.
- Create *Kitty Band*. Students are asked to create a dynamic webpage showing their personal artifacts. An example is provided by Miss Siyue Li of the University of Chinese Academy of Sciences, who created the *Kitty Band* work of **creative expression**, as shown in Fig. 2c. This witty band can play a piece of music given an input string of music scores. As a freshman of Physics major, she finished this project in 3 days. Half of her time was spent on thinking, designing, and making. Coding and debugging accounted for no more than 50% of the time.

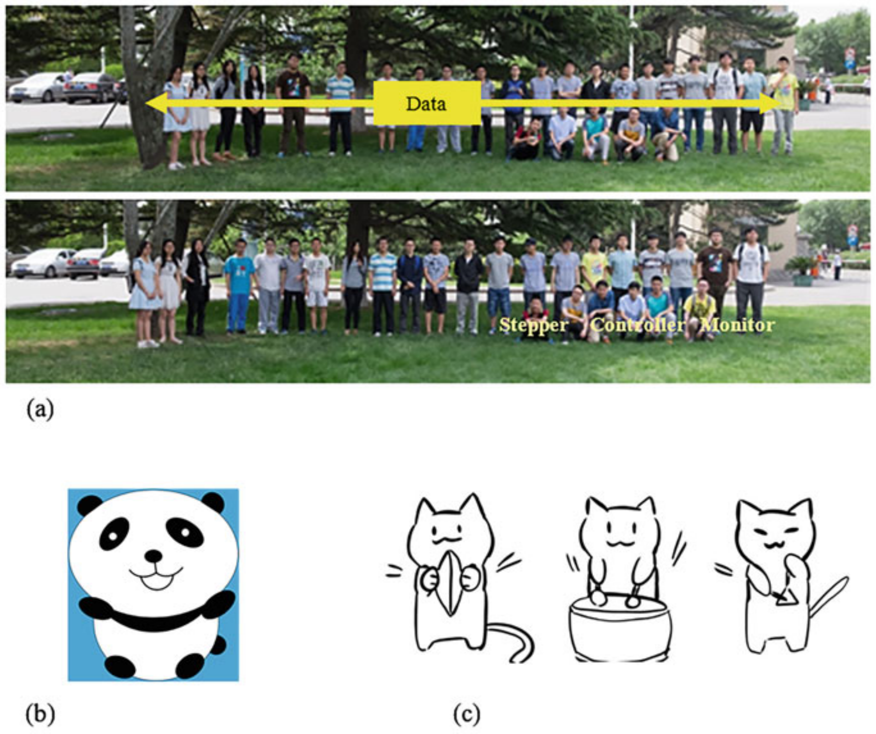


Fig. 2 Three examples of problem-solving and creative expression. (Photos and graphics credits: Haoming Qiu, Hongrui Guo, Siyue Li). (**a**) Sort a class of students: from an order by name to the order by height. (**b**) What's the area size of the panda? (**c**) Part of the *Kitty Band*

Intended Audience

The primary audience of this book are undergraduates interested in taking a Computer Science 101 (CS101) course. The material of the book has been used in the University of Chinese Academy of Sciences (UCAS) since 2014, serving a 3-credit, required course of Introduction to Computer Science for freshmen undergraduate students from the schools of Sciences, Engineering, Mathematics, Business and Management, and Arts and Humanities.

The book is also beneficial to teachers and lecturers of an Introduction to Computer Science course. The material of the book was used in two summer schools organized by China Computer Federation, to train university and high-school teachers on teaching a Computer Science 101 course utilizing computational thinking. The trainees came from all ranks of universities and top-ranking high schools.

The book is helpful to high-school students who are interested in taking a computer science advanced placement course, for instance, AP Computer Science

Principles. The contents of this book significantly overlap with the five big ideas and six computational thinking practices of AP Computer Science Principles.

For students who prefer to study by themselves, this textbook provides supplementary material and answers to even-numbered exercises. The students do need a computer to solve programming problems.

Structure of Contents

The contents of the book are organized into seven chapters and appendices. Chapters 1 and 2 introduce the computer science field. Chapters 3, 4, and 5 explain the core of computational thinking. They elaborate how logic thinking, algorithmic thinking, and systems thinking make computational problem-solving into correct, smart, and practical processes. Chapter 6 extends computational thinking to networks. Chapter 7 describes four practice projects. The project material is best used side by side with other chapters, as illustrated in Table 1.

Chapter 1 overviews the computer science field and computational thinking. It introduces the ABC features without: **A**utomatic execution, **B**it-accuracy, and **C**onstructive abstraction. It summarizes the eight understandings within: **A**utomatic execution, **C**orrectness, **U**niversality, **E**ffectiveness, comple**X**ity, **A**bstraction, **M**odularity, and **S**eamless transition. The eight understandings can be shortened to an acronym: Acu-Exams.

Note that automatic execution is a feature common to both perspectives within and without, when appreciating computer science. Chapter 1 tantalizes students with the intriguing question: Why and how trillions of instructions can be automatically executed in a fraction of a second, sometimes across the globe, to produce correct computational results? A partial answer is: abstractions in computer science are automatically executable abstractions.

The chapter also highlights the impact of computer science on society by presenting several sophisticated common senses of the field, from ICT industry to digital economy, from Chomsky's digital infinity to Boutang's bees metaphor, and from the wonder of exponentiation to wonder of cyberspace.

Chapter 2 introduces digital symbol manipulation as the core of computational processes. Simple but increasingly sophisticated examples are used to learn concepts such as numbers, characters, variables, arrays, strings, conditional, loop, von Neumann computer, processor, memory, I/O devices, instructions, etc. All of these concepts are viewed through the lens of digital symbol manipulation: data are symbols, programs are symbols, computers are symbol-manipulation systems.

Chapter 3 studies logic thinking to appreciate how to make computational processes correct. It introduces basic concepts of Boolean logic, including propositional logic and predicate logic. It introduces the Turing machine as a theoretical computer for multi-step computational processes. Church-Turing Hypothesis and Gödel's incompleteness theorems are discussed to reveal the power and limitation of computing. Accessible examples are used to explain the concepts.

Table 1 A sample course schedule for the Spring semester of the year 2020 at UCAS

Week	Lecture Two classes per week	Project Two classes per week
1	School delayed due to Covid-19	
2	CS Overview	
3	Symbol Manipulation	
4	Symbol Manipulation	
5	Logic Thinking	
6	Logic Thinking	Turing Adder
7	Logic Thinking	Turing Adder
8	Algorithmic Thinking	Turing Adder
9	Algorithmic Thinking	Text Hider
10	Algorithmic Thinking	Text Hider
11	Holiday break	
12	Midterm Review	Text Hider
13	Systems Thinking	Human Sorter
14	Systems Thinking	Human Sorter
15	Systems Thinking	Human Sorter
16	Network Thinking	Web Artifact
17	Network Thinking	Web Artifact
18	Network Thinking	Web Artifact
19	Term Review	
20	Final Exam	

Chapter 4 studies algorithmic thinking to appreciate how to make computational processes smart. This includes smart ways to define, measure, design, and adapt algorithms. After introducing the basic concepts of algorithm and algorithmic complexity, this chapter uses some examples to explain the design and analysis of algorithms. Discussed algorithmic concepts include divide-and-conquer, dynamic programming, the greedy approach, randomization, hashing, sort, search, algorithmic complexity, and P versus NP.

Chapter 5 studies how systems thinking makes computational processes practical, by discussing three key concepts: abstraction, modularization, and seamless transition. Elementary data abstractions and control abstractions are discussed here in one place. Hardware and software concepts are introduced as systems modules in increasing abstraction levels, from logic gates and memorizing devices, combinational circuits, sequential circuits, to instruction pipelines and software stack. This chapter also discusses four "laws" that make seamless execution possible: Yang's cycle principle, Postel's robustness principle, von Neumann's exhaustiveness principle, and Amdahl's law.

Chapter 6 extends computational thinking to networks, including the Internet and the network of webpages. Two main knowledge thrusts, connectivity and protocol stack, are discussed to introduce concepts and methods such as naming, topology, packet switching, TCP/IP protocols, DNS, WWW, viral marketing, Metcalfe's law, and responsible computing.

Chapter 7 describes four practice projects which are an integral part of the course. They are inspired by the US National Research Council's characterization: "computer science is the study of ... *abstract computers*, ... *real computers*, ... and *applications of computers*." The Turing Adder project augments students' understanding of abstract computer. The Human Sorter project invites students to design a real computer. The Text Hider project represents a computer application. Finally, the Personal Artifact project offers students an opportunity to demonstrate their capability of creative expression, by creating a dynamic webpage.

This chapter also reviews responsible computing, including code of conduct and best practices for independent work, collaboration, and acknowledgment.

How to Use This Book?

Teaching and learning an introductory course of computer science must balance two facts about the student community. First, many students do not have prior experience in computer science. We polled the 2014–2018 classes of the CS101 course at the University of Chinese Academy of Sciences, where each year there were about 340–390 students in the class. The results show that over 90% of students had no prior experience in CS or programming. For the 2014 and 2015 classes, over 6% of students did not own a personal computer when they came to the university. We need to make sure inexperienced but hard-working students can earn good grades.

Second, most students, both experienced and inexperienced, do get the hang of introductory computer science quickly. We need to ensure that students still find CS101 intellectually interesting, not a watered-down, boring course.

Based on our 6-year teaching experience, we offer the following suggestions:

- Normal learning with contents augmented by Bloom's taxonomy.
- Utilizing Knuth's Test to instantiate Bloom's taxonomy for CS101.
- Focusing on the elementary and leaving space for experienced students.

This textbook can be used in a CS101 course in the normal way, with lectures, homework exercises, projects, and exams. Some lecturers and students may find that this textbook contains a lot of material for mind-active and hands-on learning. That is, the book aims at the upper levels of Bloom's taxonomy.

Shown in Fig. 1, Bloom's taxonomy is a taxonomy of educational objectives first proposed in 1956 and revised in 2001. It organizes six levels of educational objectives into a pedagogic pyramid. We find that it is feasible and desirable to aim at higher levels of Bloom's taxonomy in a CS101 course.

A significant portion of this book is designed to enable lecturers and students to rise from the basement level of "remember" to the top level of "create" in Bloom's taxonomy. For instance, after learning an adder, students may be asked to design a never-discussed subtractor. The knowledge and capability needed are beyond simply memorizing. The Personal Artifact project asks a student to independently create a dynamic webpage by the end of the semester. In doing so, the students learn how to turn personal insights and creative ideas into computational artifacts. The book provides a library of dozens of webpages created by past students and teaching assistants. Students are enabled to create their webpages, learning by themselves Web programming along the way, including the needed HTML, CSS, and JavaScript knowledge, as well as proper code of conduct.

It was not until the Spring semester of the year 2020 that we realized that the learning method we have practiced for 6 years can be summarized in one sentence: utilize Knuth's Test to instantiate Bloom's taxonomy.

In an interview in February 2020, Donald Knuth stated beautifully an instantiation of the "create" level in Bloom's taxonomy for computer science education:

> The ultimate test of whether I understand something is if I can explain it to a computer. I can say something to you and you'll nod your head, but I'm not sure that I explained it well. But the computer doesn't nod its head. It repeats back exactly what I tell it. In most of life, you can bluff, but not with computers.

We call this "ultimate test" **Knuth's Test**. It offers students a pedagogic tool to check if they have learned a unit of knowledge or capability: see if they can explain it to a computer. Running a program on a PC is an obvious way to perform Knuth's Test. Executing a computational process on a human-computer as a thought experiment is another way. A student cannot bluff with either type of computer.

The above-suggested practice of teaching and learning could go out of hand, by exposing students to too much material. Thus, we have the third suggestion: focusing on the elementary and leaving space for experienced students. The contents of the book have been purposely designed to focus on the elementary of computational thinking, such that the material can be covered in full in one semester. Material targeting experienced or hungry students is explicitly marked.

For instance, although dozens of programming examples and exercises are included, a student is required to write only 300 lines of code for Go programming. The emphasis is on general ideas and methods of programming, not on Go-specific syntax and semantics. When facing the new task of creating a dynamic webpage, most students can quickly learn Web programming by themselves.

Suggested schedules for a 3-credit, 60-period course, and a 2-credit, 40-period course are shown in Tables 2 and 3, respectively. Note that 40% of class time is devoted to the projects for the 3-credit course, and 30% for the 2-credit course. A homework assignment is handed out for each of the first six chapters.

Due to Covid-19, we had to conduct CS101 as an online course for the Spring semester of 2020 (Table 1). The students did fine, comparing to previous classes. However, the working time of lecturers and TAs increased by 40%. This was mainly due to first-time overheads. Future online courses could be more efficient.

Table 2 Suggested schedule for a 3-credit course

Week	Lecture Two classes per week	Project Two classes per week	Due date 23:30 pm, Sunday
1	CS Overview		
2	Symbol Manipulation		Homework 1
3	Symbol Manipulation		Homework 2
4	Logic Thinking		
5	Logic Thinking	Turing Adder	
6	Logic Thinking	Turing Adder	Homework 3
7	Algorithmic Thinking	Turing Adder	Project 1
8	Algorithmic Thinking	Text Hider	
9	Algorithmic Thinking	Text Hider	Homework 4
10	Midterm Review	Text Hider	Project 2
11	Systems Thinking	Human Sorter	
12	Systems Thinking	Human Sorter	Homework 5
13	Systems Thinking	Human Sorter	Project 3
14	Network Thinking		
15	Network Thinking	Web Artifact	
16	Network Thinking	Web Artifact	Homework 6
17	Term Review	Web Artifact	Project 4
18	Final Exam		

Table 3 Suggested schedule for a 2-credit course

Week	Lecture Two classes per week	Project Two classes per week	Due date 23:30 pm, Sunday
1	CS Overview		
2	Symbol Manipulation		Homework 1
3	Symbol Manipulation		Homework 2
4	Logic Thinking		
5	Logic Thinking	Turing Adder	Homework 3
6	Algorithmic Thinking	Turing Adder	Project 1
7	Algorithmic Thinking	Text Hider	Homework 4
8	Midterm Review	Text Hider	Project 2
9	Systems Thinking		
10	Systems Thinking	Human Sorter	Homework 5
11	Network Thinking	Human Sorter	Project 3
12	Network Thinking		Homework 6
13	Term Review		
14	Final Exam		

Notations

Some widespread programming notations are used in this book: the camel notation, the dot notation, the slash notation, the quotation marks, and notations for hexadecimal and Unicode values.

The **camel notation** is also called the camel case notation. It is used to denote various names (e.g., variable or file names), such as MyPicture, studentsMap, and doctoredAutumn. This practice writes the phrase of a name together with the first letter of each word capitalized, resembling the humps of a camel. The first word may all be in a small case.

Students may have already seen the dot notation used as a file extension, such as myHW2.pdf, or in Web domain names such as www.ucas.edu.cn. The **dot notation** is also used to denote the component of a program construct, such as the member of a struct variable or the function in a program package. For instance, the notation

```
fmt.Println
```

calls the Println function in the fmt package. The dot notation

```
A.Key
```

refers to the key component in variable A, which has a data type of struct.

The slash (/) notation is used mainly to denote the path name of a file. For instance, the following **slash notation**

```
/cs101/Prj2/ucas.bmp
```

denotes the full path name of a file, where the first slash denotes the root directory, followed by the cs101 subdirectory, followed by the Prj2 sub-subdirectory, followed by the real file ucas.bmp. The four entities are separated by three slashes.

The single **quotation marks** denote a character, e.g., ‘A’, ‘6’, and ‘?’. The double quotation marks denote a character string, e.g., “Alan Turing”.

The 0x and 0X notations are used to denote hexadecimal numbers, such as 0x36, 0x1f, and 0X1F. Some programming systems differentiate these two notations for a small case and a capital case. We do not differentiate them unless required.

The U+ notation denotes a Unicode value. For instance, the Chinese character ‘志’ and the Euro sign ‘€’ have Unicode encoding values of U+5FD7 and U+20AC, respectively.

A 3-star notation, ‘***’, is used to mark material targeting experienced and hungry students. Material for all students has no marking.

An **Example** ends with the notation ☶, the trigram symbol for the mountain in *Book of Change*, which symbolizes “the end”. The following is an instance.

Example 1. $\mathbf{(110.101)_2 = (?)_{10}}$

$(110.101)_2 = 1\times2^2+1\times2^1+0\times2^1+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}=4+2+0.5+0.125$
$= (6.625)_{10}.$

Supplementary Material

The companion website **cs101.ucas.edu.cn** provides supplementary material for (1) lecture and projects slides, (2) the source code of all programs, and (3) solutions to even-numbered homework exercises, as well as other teaching and learning aids.

Acknowledgments

We are grateful to many people for feedback and encouragement. Students of classes 2014–2019 at the University of Chinese Academy of Sciences were the first batch of practitioners of this book's material. We are happy to see that over 90% of graduates of these classes go on to pursue advanced degrees.

We are indebted to Professor Donald Knuth of Stanford University for his fundamental inspiration. We adopt his recent advice, called Knuth's Test in this book, as a pedagogic tool. We thank Professor Jeanette Wing of Columbia University for explaining her view on computational thinking. We are grateful to Professor Xiaomeng Xu of Idaho State University for introducing us to Bloom's taxonomy.

Professor Xiaoming Li of Peking University has provided persistent encouragement and feedback. Professors Guoliang Chen and Lian Li, chairs of the Education Steer Committee on Computer Fundamentals of China's Ministry of Education, tirelessly lead the computational thinking reform in China for over 10 years. The many workshops they chaired helped the design and development of the book's material. Professor Xiaoming Sun of the University of Chinese Academy of Sciences is a main designer of the CS101 course at UCAS. Hongrui Guo and Zishu Yu, our teaching assistants, have helped develop the material of the project.

This book uses material from several institutions, including company names, product names, logos, and images. We acknowledge that all such material is the property of the owner. All images are reproduced with permissions. The institutions include ACM, Amazon.com, AMD, Apple, AT&T, Baidu, China Computer Federation (CCF), Cisco, the College Board, Facebook, Google, RedHat, Huawei, IBM, IEEE-Computer Society, Intel, Lenovo, LinkedIn, Microsoft, Oracle, Sugon, Tencent, the World Wide Web Consortium (W3C), and Xiaomi. ACM and IEEE-CS are international societies of computer professionals. CCF is the society of computer professionals in China with over 60,000 members.

Special thanks are due to the open-source software community. This book uses the following open-source software: the Linux operating system, the Go programming language, the VirtualBox tool, the Visual Studio Code (VSCode) editor, and Web server and browser software.

We acknowledge the support of the Innovation Institute of Network Computing, Chinese Academy of Science, where research and education are integrated to enable innovation.

We thank Dr. Celine Chang, Veena Perumal, and Suganthi Tamijarassou of Springer Nature for making a smooth publication process.

Bibliographic Notes

Quotations from Donald Knuth are from an interview in February of 2020 by *Quanta Magazine* [1]. Bloom's taxonomy of educational objectives was presented in [2] and updated in [3]. Computational thinking is discussed in [4–6]. Different ways to introduce computer science are presented in [7–11].

References

1. D'Agostino S (2020) The computer scientist who can't stop telling stories. Quanta Magazine. https://www.quantamagazine.org/computer-scientist-donald-knuth-cant-stop-telling-stories-20200416
2. Bloom BS, Engelhart MD, Furst EJ et al (1956) Taxonomy of educational objectives: Cognitive domain. Longman Group
3. Anderson LW, Krathwohl DR, Airasian PW, Cruikshank KA, Mayer RE, Pintrich PR, Raths J, Wittrock MC (2001) A taxonomy for learning, teaching, and assessing: a revision of Bloom's taxonomy of educational objectives, abridged edition. Longman, White Plains
4. Wing JM (2006) Computational thinking. Commun ACM 49(3):33–35
5. US National Research Council (2004) Computer science: reflections on the field, reflections from the field. National Academies Press, Washington, DC
6. College Board (2020) AP computer science principles course and exam description. https://apcentral.collegeboard.org/pdf/ap-computer-science-principles-course-and-exam-description.pdf
7. Page D, Smart N (2014) What is computer science?: An information security perspective. Springer, Switzerland
8. Schneider GM, Gersting J (2018) Invitation to computer science (8th edn). Cengage Learning, Boston
9. Dale N, Lewis J (2019) Computer science illuminated (7th edn). Jones & Bartlett Learning, Burlington

10. Brookshear G, Brylow D (2019) Computer science: an overview (13th edn). Pearson, London
11. Alvarado C, Dodds Z, Kuenning G, Libeskind-Hadas R (2019) CS for all: an introduction to computer science using Python. Franklin, Beedle & Associates Inc., Wilsonville

Contents

Chapter 1
Overview of Computer Science

The ultimate test of whether I understand something is if I can explain it to a computer. . . . In most of life, you can bluff, but not with computers.
—Donald Knuth, 2020

Computer science is an academic discipline that studies computational processes in solving problems in scientific, engineering, economic and social domains. Computational thinking is the way of thinking by computer scientists, which underlies the bodies of knowledge in the computer science discipline. Computer science provides an intellectual foundation supporting the information technology industry, the worldwide digital economy and the information society. It exhibits three wonders and three persuasions while permeating modern civilizations.

In this chapter, students will see and use a number of small computer programs. They will start writing programs in Chap. 2.

1.1 Computational Processes in Problem Solving

Computer science studies computational processes, i.e., processes of information transformation. It differs from fields of natural sciences such as Physics, Chemistry, or Biology, which mainly study processes of matter and energy transformations.

A **computational process** is a problem-solving process of information transformation, via a sequence of digital symbol manipulation steps. Computational processes often manifest as automatic executions of programs on computer systems.

A binary digit (**bit**) takes on a value of 0 or 1. A **digital symbol** is any notation that is representable as one or more bits, to denote any concrete or abstract entity. **Manipulation** is a sequence of operation steps on digital symbols, where the length of the sequence can be one or many. Operation steps are also called **operations**.

An **algorithm** is a finite set of rules specifying a sequence of operations on digital symbols to solve a problem. A **program** is an expression of an algorithm in a computer language, such as the Go programming language. A program segment,

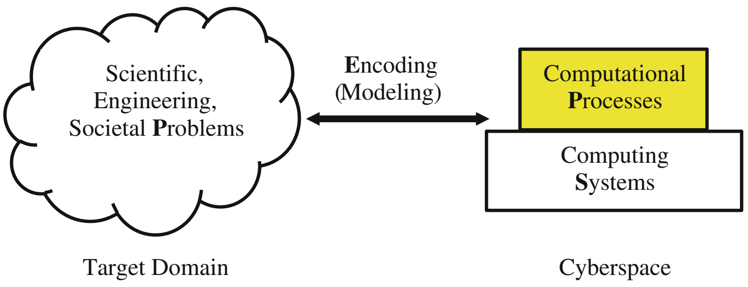


Fig. 1.1 Computational processes in problem solving: the PEPS model

part or whole, is called **code**. A group of digital symbols is called **data**. An algorithm often produces output data from input data.

A program is expressed in a programming language as a group of digital symbols. Thus, programs can be viewed as data. When programs and data are stored in a computer in a non-volatile way (i.e., data still exist even when the power is turned off), they are called **files**. We store program files and data files in a computer.

As illustrated in Fig. 1.1, a computational process in problem solving involves four aspects, abbreviated as **PEPS** for **P**roblem, **E**ncoding, Computation **P**rocess, and Computer **S**ystem. **Cyberspace** refers to the right part of Fig. 1.1, namely, computer systems plus the computational processes executing on them.

- **Problem**. We study computational processes to solve problems in target domains, i.e., fields of applications. Computer science can be used to help solve problems in many fields, including mathematics, natural sciences, social sciences, engineering and technology, economics and business, and even arts and humanities fields. We will elaborate why computer science permeates when discussing digital infinity and computational lens in Sect. 1.3.
- **Encoding**. Domain problems are converted to computational problems to be solved in the cyberspace, which consists of computational processes automatically executing on computer systems. This converting process is called **encoding** or **modeling**, often done by humans. For a specific domain problem, encoding generates a computational problem and an expected computational solution, manifesting as a model of the problem in cyberspace and an algorithm to solve the problem. Encoding often determines the accuracy and precision of the solution. Note that the encoding process is actually a bidirectional process. It is common practice that humans are ultimately responsible for converting the domain problem in the target domain to the computational problem in the cyberspace, and then converting the solution in the cyberspace back to the solution in the target domain. There is much opportunity for human's imagination and creativity to play out in this bidirectional mapping, including formulating the problems, designing approaches and solutions, deciding human-computer symbiosis and interaction, and iterative optimization.

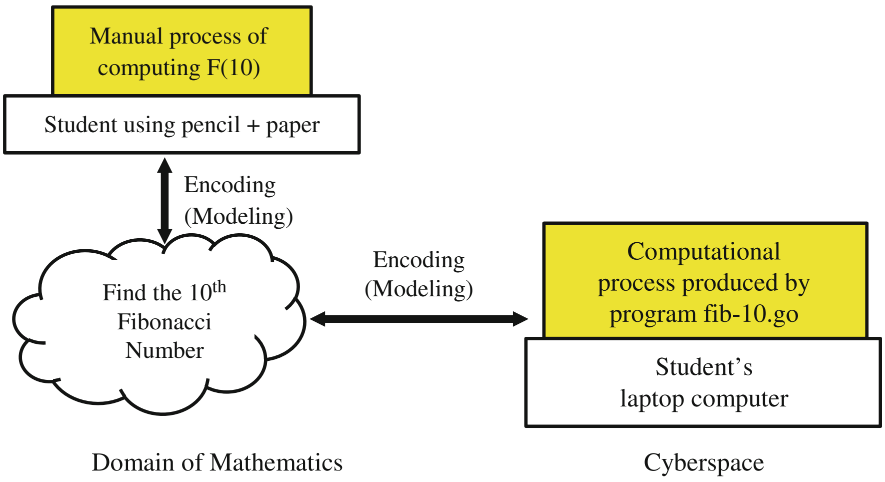

Manual process of computing F(10)
Student using pencil + paper
Encoding (Modeling)
Find the 10th Fibonacci Number
Encoding (Modeling)
Computational process produced by program fib-10.go
Student's laptop computer
Domain of Mathematics
Cyberspace

Fig. 1.2 Two processes for computing the 10th Fibonacci number F(10)

- Computational **Process**. A computational process often manifests as a running computer program, which embodies the human designed model and algorithm to solve the problem. The program specifies the computational process of information transformation via step-by-step digital symbol manipulations. **Programming** is the activity to design and develop a program. To obtain the final effective and efficient computational process, we may need many iterations of encoding, programming and execution, even when the underlying computer system is given. Encoding, designing, and programming can be combined into one process in practice, especially when the problem is simple or small.
- Computer **System**. The computer system may be in many forms, abstract or real. The examples, exercises and practice projects in this book mostly use two types of real computer systems: the student's laptop computer and the World Wide Web. The Human Sorter project creates a real computer consisting of humans.

Example 1.1. Computing a Small Fibonacci Number
A problem can be solved by different encodings. Figure 1.2 illustrates two processes in computing the 10th Fibonacci number F(10): one by manual computing and the other by a computer program. Contrasting these processes highlights the importance of automatically executed computational processes.

Problem. The problem is to find the 10th Fibonacci number F(10) in the domain of mathematics. Note that the mathematical definition of Fibonacci numbers is:

F(0)=0, F(1)=1; F(n)=F(n-1)+F(n-2) when n>1.

A student may use the mathematical definition to manually compute the first 11 Fibonacci numbers using a pen and paper. Given F(0)=0, F(1)=1, one has

F(2)=F(1)+F(0)=1+0=1,
F(3)=F(2)+F(1)=1+1=2,

```
Output F(10)                // n and F(n) are natural numbers
where F(n) is defined as
      if (n=0 or n=1) then F(n)=n else F(n)=F(n-1)+F(n-2)
```

(a)

```
package main                                  // Program setup
import "fmt"
func main() {
    fmt.Println("F(10)=", fibonacci(10))      // Output F(10)
}
func fibonacci(n int) int {                   // fibonacci(10)
    if n == 0 || n == 1 {                     // If n=0 OR n=1, (|| means OR)
        return n                              //     return n and exit
    }                                         // Recursively call
    return fibonacci(n-1)+fibonacci(n-2)      //   fibonacci(9) and fibonacci(8)
}
```

(b)

```
> go build fib  -10.go
> ./fib-10
F(10)= 55
>
```

(c)

Fig. 1.3 Computational process for finding the 10th Fibonacci number F(10). Texts after a double slash (//) are **comments** to explain the code. (**a**) An algorithm to find F(10) directly from the mathematical definition. (**b**) A Go program fib-10.go that implements the algorithm. (**c**) Compile fib-10.go and execute fib-10 to produce the output

F(4)=F(3)+F(2)=2+1=3,
F(5)=F(4)+F(3)=3+2=5,
F(6)=F(5)+F(4)=5+3=8,
F(7)=F(6)+F(5)=8+5=13,
F(8)=F(7)+F(6)=13+8=21,
F(9)=F(8)+F(7)=21+13=34,
F(10)=F(9)+F(8)=34+21=55.

This manual calculation process is tedious and time consuming. For a small n, e.g., n=10, a student may manually compute F(n) in a few seconds or minutes. But how about finding F(50) or F(5000000000)? Fortunately, step-by-step computational processes that are tedious and time consuming for humans are often good candidates for computer processing. Figure 1.2 shows another process for computing F(n), which is a process of information transformation via step-by-step digital symbol manipulations. This cyberspace solution is further elaborated in Fig. 1.3.

Encoding. In the cyberspace, the problem is to compute F(10) automatically by a computer, not by manual calculation. Its solution is encoded as a recursive algorithm directly from the mathematical definition, as shown in Fig. 1.3a. It is a *recursive* algorithm because the function F calls itself recursively in F(n)=F(n-1)+F(n-2). Note that this recursive algorithm in cyberspace is different from the algorithm of the manual calculation process in the mathematics domain.

Computational Process. The computational process is embodied in the Go program of Fig. 1.3b, which implements the algorithm in Fig. 1.3a. This is a straightforward implementation, almost literally copying Fig. 1.3a into the Go programming language syntax. Each line of the program code is called a **statement**. Recall that we use **code** to refer to a segment of a program. The first three statements are to set up the program. The function name "F" is replaced by a longer but more informative name "fibonacci". The statement

```
fmt.Println("F(10)=", fibonacci(10))
```

is to Output F(10), that is, to print out the result value of F(10). The next 6 lines of code form a subprogram, called a **function**, which does the actual computation of Fibonacci numbers. Given an integer n as the input parameter, the function generates an integer output fibonacci(n) by implementing the algorithm in Fig. 1.3a.

The computer screen outputs are shown in Fig. 1.3c. To summarize, the actions of encoding, programming, and entering commands are done by the human user, but actual compilation and program execution are done by the computer. This way of dividing labor is called *human-computer symbiosis*.

- Human: convert the math problem to the Go program fib-10.go.
- Human: enter the compile command "go build fib-10.go".
- Computer: execute command "go build fib-10.go", to compile the **high-level language program** file fib-10.go into an executable program file fib-10. **Compilation** refers to converting a high-level language program to an executable program, also called a **machine code** program.
- Human: enter the program execution command "./fib-10".
- Computer: execute command "./fib-10", to execute program fib-10 and produce screen output "F(10)=55".

Computer System. In this example, the computer is the student's laptop computer supporting the Go programming language and the Linux operating system.

The above simple example already reveals the rich meaning of the concepts of "computational process in problem solving", as well as of "step-by-step digital symbol manipulation". After encoding, the mathematical problem is converted into a computational problem and a solution. The algorithm in Fig. 1.3a, the Go program in Fig. 1.3b, and the compilation and execution processes in Fig. 1.3c, all represent processes of information transformation via digital symbol manipulations.

It is obvious that the final result F(10)= 55 is a combination of digital symbols. It may not be as obvious that the Go program fib-10.go and the executable program file fib-10 are also digital symbols. Manipulation operations include steps of

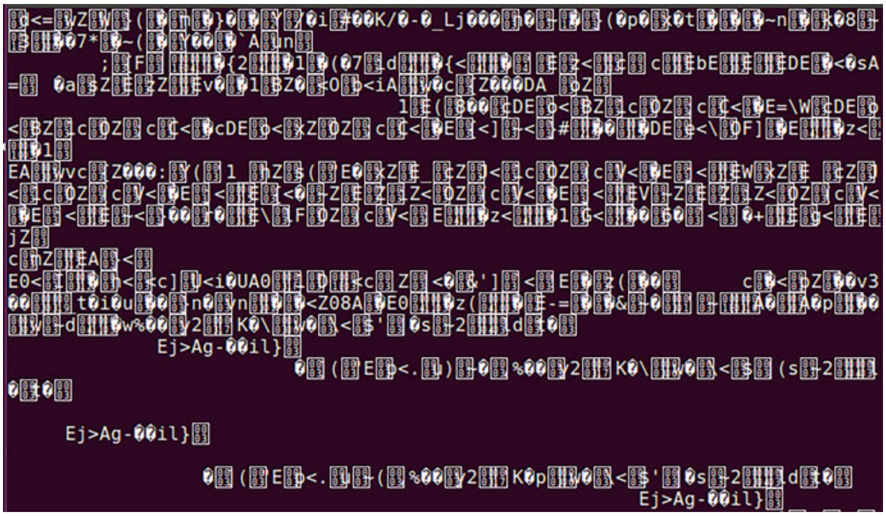


Fig. 1.4 Screen display of the machine code fib-10: scrambled symbols

programming, compilation, machine code execution, as well as more detailed operations described inside the Go program of Fig. 1.3b.

Why do we go this roundabout way of (1) writing a program fib-10.go, (2) compiling fib-10.go into fib-10, and (3) executing fib-10? Why don't we simply write and execute fib-10?

The computer only understands and executes a machine code program, such as fib-10, which consists of a sequence of 0's and 1's. When displaying fib-10 on the computer screen, one sees the scrambled result shown in Fig. 1.4. It is difficult for human to understand a machine code program such as fib-10. For this reason, a machine code program such as fib-10 is also called a **low-level language program**.

It is easier for the human to understand a high-level language program. However, the computer cannot directly understand and execute a high-level language program, such as fib-10.go. A compiler is needed to convert a high-level language program into a machine code program that is directly executable on a machine.

During this compilation process, the compiler also checks for and reports various **compile-time errors**, such as syntactic errors in the high-level language program fib-10.go. However, **runtime errors** may still exist in the compiled machine code fib-10, even when no error is reported during the compilation process. **Bugs** are the term used to refer to all errors of a program, including compile-time errors and runtime errors.

Refer to Fig. 1.3c. The command "go build fib-10.go" directly execute on a computer but looks like a high-level language statement. In fact, commands are high-level language programs called *shell scripts*. What happens is that when human enters a command, a software tool, called **interpreter**, works behind the scene to automatically interpret (i.e., convert the command into machine code and then execute, one statement at a time). The computer actually executes machine code of

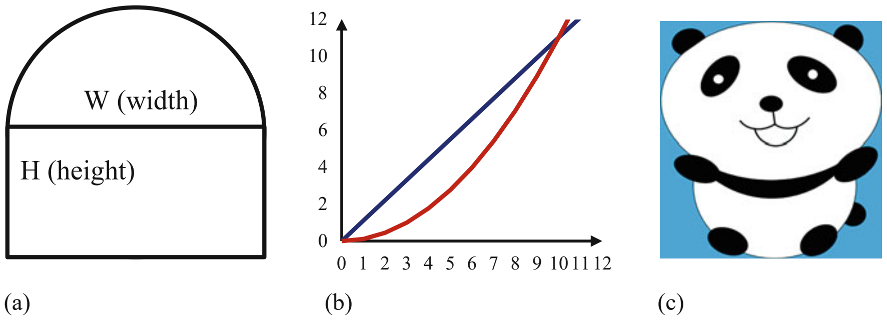


Fig. 1.5 Computer science enables us to compute sizes of irregular shapes. (**a**) School Mathematics Regular shapes. (**b**) College Mathematics Curly shapes. (**c**) Computer Science Irregular shapes

the command, not the command itself. An operating system such as Linux normally provides a command interpreter called **shell**.

Computing is more than automatic execution of arithmetic operations. Three cases below are used to demonstrate the power and beauty of computational processes in augmenting other disciplines, showing that computational thinking can change the way of thinking in solving problems by bringing in new values.

Step-by-step computing is powerful. The basic idea in Example 1.1 looks trivial: step by step computing of the sequence of Fibonacci numbers. However, Fibonacci sequence was a key innovative idea enabling scientists to solve Hilbert's 10th problem, an important mathematics problem asked by David Hilbert in 1900. The problem is to find an algorithm to determine whether any given Diophantine equation has an integer solution. The answer, provided in 1970, is No. It is interesting to note that this 70-year work is a multidisciplinary research, where the main result is called the MRDP Theorem, after four people: Yuri Matiyasevich, a Russian mathematician; Julia Robinson, the first female President of the American Mathematical Society; Martin Davis, a computer scientist; and Hilary Putnam, a past President of the American Philosophical Association.

Augment the problem. Computational thinking can extend the scope of problems, enabling people to solve problems traditionally intractable. A case in point is to compute the area of an irregular shape. Mathematics in primary and high schools enable students to compute the area of a regular shape enclosed by straight lines and circles. College Mathematics goes further by enabling students to compute the area of a curly shape enclosed by curves of two or more functions. For instance, the size of the area in Fig. 1.5a is the sum of a rectangle and a semicircle, which is $W \cdot H + (W/2)^2 \cdot \pi/2$. The size of the area enclosed by the straight line $y = 1.1 \cdot x$ and the square curve $y = 0.11 \cdot x^2$ in Fig. 1.5b is $\int_0^{10}(1.1x - 0.11x^2)dx$.

However, such school or college math is inadequate to handle the task of computing the area of the panda picture in Fig. 1.5c, which is an example of irregular shapes. Computer science offers new capabilities to routinely compute such irregular areas. A specific method called Monte Carlo simulation is shown in the Personal

Artifact project. The same idea can extend to multiple dimensions, and can be used to compute volume, mass, energy, number of particles, etc.

Change approaches to the problem. Computational thinking can inspire radically new approaches to domain problems. A case in point is Human Whole-Genome Shotgun Sequencing. The complete sequencing of the human genome is a landmark endeavor in biology and health science. In 1990, the United States government officially started the Human Genome Project (HGP), and later set the goal of sequencing the human genome by 2005 at US$1 per chemical base pair. That is, the total dollar and time costs would be $3 billion and 15 years. However, by 1998, only 5% of the human genome were sequenced.

In 1997, Gene Myers and Jim Weber proposed to attack the human genome sequencing problem by a radical approach, called Whole-Genome Shotgun Sequencing. The idea is to break down the DNA sequence into random fragments, sequence those fragments, and then assemble them in the correct genome order. This approach was used to successfully sequence the genome of H. influenzae bacterium of 1.8 million base pairs. Myers and Weber projected the approach could apply to the much larger human genome, because we could heavily utilize effective algorithms and much faster computing technology. The established community rejected their proposal, judging the method would fail for the human genome with 3 billion base pairs.

In 1998 Myers joined a newly founded company called Celera Genomics to realize his computation-heavy Human Whole-Genome Shotgun Sequencing approach. His team developed new algorithms and more than 500 thousand lines of code for a 7000-processor parallel computer. This approach proved to be effective. In 9 months from September 1999 to June 2000, Celera finished a rough draft sequence of the human genome. On June 26, 2000, Celera joined other scientists, US President Bill Clinton and British Prime Minister Tony Blair to announce the completion of an initial sequencing of the human genome.

1.2 Characteristics of Computational Thinking

Computational thinking is the way of thinking by computer scientists, which underlies and manifests as the bodies of knowledge in the computer science discipline. When viewed from the perspective of a way of thinking, computer science and computational thinking are synonymous.

Computational thinking can be characterized from three angles: (1) the three features without, (2) the eight understandings within, and (3) the research view of computer science. They are not separate things but different perspectives. Computational thinking is the synergy of all these perspectives, similar to a symphony played on multiple music instruments.

1.2.1 The Three Features Without

When viewed from the outside, namely, from the computer user's perspective, computer science exhibits three features distinct from other fields, called the **ABC features**: **A**utomatic execution, **B**it accuracy, and **C**onstructive abstraction. The ABC features are listed in Table 1.1 with examples and counterexamples.

Automatic execution is easy to understand. Computer science targets those bodies of knowledge (whether they are theory, hardware, or software) which enable computational processes to be automatically executed on computers. That is why computer science emphasizes exact, step-by-step processes. Only such processes can be understood by computers, thus amiable to mechanic, step-by-step automatic execution. Even for human-in-the-loop processes, computational thinking will try to make them largely automatic and seamless.

This feature can be seen by comparing the two scenarios in Table 1.1: (1) computing the 10th Fibonacci number F(10) by human using pen and paper, where each step needs human to manually operate; and (2) computing F(10) by running the fib-10.go program, where the computational process is executed automatically on a computer. The second scenario is much faster, especially when the problem is to compute a larger Fibonacci number such as F(50), F(5000) or F(5000000).

Example 1.2. Computing Larger Fibonacci Numbers F(50) and F(100)
The manual calculation process of Fibonacci numbers in Example 1.1 is tedious and slow. This becomes obvious if students are asked to manually compute a larger Fibonacci number, such as to compute F(50). In contrast, automatic execution on a computer allows us to compute F(50) easily. We only need to slightly modify fib-10.go by changing 10 to 50, and then compile fib-50.go and execute, to get F(50)= 12586269025 in a few minutes. How about computing F(100)? Repeat the above programming-compilation-execution processes by changing 10 to 100. This will reveal two caveats of automatically executed computational processes: an apparently

Table 1.1 The ABC features at a glance

Feature	Example	Counter example
Automatic execution of a computational process on a computer	Computing the 10th Fibonacci number by running fib-10.go	Computing the 10th Fibonacci number by human using pen and paper
Bit accuracy: a computational process is accurate to every bit	Processing scientific experimental data by a computer	An experiment judged by statistically significant result (P-value <0.05)
Constructive abstraction: to form a general entity from individual instances by smartly composing a group of more primitive entities	The von Neumann model abstracting many real computers. A program to find Fibonacci numbers by dynamic programming	A human's feeling of happiness A damaged binary code file for the same Go program, consisting of gibberish bits

correct computational process could (1) become terribly slow and (2) produce incorrect results. The moral: *being automatic is not enough*.

Bit accuracy is also intuitive. Any scientific field needs its academic rigor by pursuing accuracy and precision. Computer science pursues *bit accuracy*: any computational process is accurate and precise up to every bit. Here **bit** is short for *binary digit*, the smallest digital symbol which has a value of 0 or 1.

A counterexample of bit-accuracy is shown in Table 1.1. Scientific experiments have requirements of accuracy and precision according to the standards and best practices of their domains. For instance, we may see expressions such as "experiments results are statistically significant when the p-value is less than 0.05", "the error is no more than 3 Angstrom (**Å**)", and "the results are precise up to four digits after the decimal point". All of these are not bit accurate.

Computer science works complementarily with these domains by guaranteeing bit accuracy when processing experimental data, doing simulation, or conducting theoretical reasoning, while each domain uses its own degree of accuracy and precision. In other words, bit accuracy and domain accuracy work hand in hand.

Example 1.3. Using Binet's Formula to Compute Larger Fibonacci Numbers
We can use a closed form mathematical formula to make the computation of F(n) faster. We utilize the so called Binet's formula $F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. Note that this formula involves real numbers such as $\sqrt{5}$ and $\frac{1+\sqrt{5}}{2}$. Let us use this formula to compute F(50), F(100), and F(500), and see what happens. The revised computational process using a new program fib.binet-50.go is shown in Fig. 1.6.

This fib.binet-50.go program can be automatically executed, and is indeed much faster than fibonacci-50.go. However, it does not produce the exact integer results, but only approximate results represented as real numbers. To easily see the differences, we list below the exact integer results and the corresponding approximate results for F(50), F(100), and F(500). Exact results are in boldface.

F(50)	= 1.2586269024999998e+10	= 1258 6269 024.9 99998
F(50)		= **1258 6269 025**
F(100)	= 3.542 2484 8179 2618 e+20	= 3542 2484 8179 2618 00000
F(100)		= **3542 2484 8179 2619 15075**
F(500)	= 1.3942322456169767e+104	
F(500)	= 1394 2322 4561 6976 7000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 00000	
F(500)	= **1394 2322 4561 6978 8013 9724 3828 7040 7283 9500 7025 6587 6973 0726 4108 9629 4832 5571 6228 6329 0691 5576 5887 6222 5212 94125**	

```
Output F(50)            // n and F(n) are real numbers
```
where $F(n) = \frac{\varphi^n-(1-\varphi)^n}{\sqrt{5}}$, and $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

(a)

```
package main
import "fmt"
import "math"                    // utilize the math library
func main() {
  fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) float64 {
  sqrt5 := math.Sqrt(5)          // assign the square root of 5 to sqrt5
  phi := (1+sqrt5)/2             // assign the golden ratio to phi
  return  (math.Pow(phi,float64(n))-math.Pow((1-phi),float64(n)))/sqrt5
}
```
(b)

```
> go build fib.binet-50.go
> ./ fib.binet-50
F(50)= 1.2586269024999998e+10
>
```
(c)

Fig. 1.6 Using Binet's formula to compute the 50th Fibonacci number F(50). (**a**) An algorithm to find F(50) directly from Binet's formula. (**b**) A Go program fib.binet-50.go that implements the algorithm. (**c**) Compile fib.binet-50.go and execute fib.binet-50 to produce the output

Since Binet's formula involves real numbers, the fib.binet-50.go program in Fig. 1.6 utilizes 64-bit floating-point numbers (computer representation of real numbers) and the system-provided "math" library. Given an integer number n as input, the fibonacci function returns a 64-bit floating-point number as output.

The value of the returned number is $\frac{\varphi^n-(1-\varphi)^n}{\sqrt{5}}$, which in Go notation is:

(math.Pow(phi,float64(n))-math.Pow((1-phi),float64(n)))/math.Sqrt(5)

in Fig. 1.6b, where the power function math.Pow(a, b) returns value a^b, and float64 (n) returns value of integer n in 64-bit floating-point number representation.

Program fib.binet-50.go computes Fibonacci numbers using floating-point numbers. Some precision is lost during the problem encoding stage. But the fib.binet-50. go program itself is still bit accurate in that all operations are accurate up to every bit of the floating-point numbers involved. For instance, F(100)=F(99)+F(98) is computed as follows.

F(98)= 1.353 0185 2344 7067 e+20 = 1353 0185 2344 7067 00000
F(99)= 2.189 2299 5834 55514 e+20= 2189 2299 5834 5551 40000
F(100)=3.542 2484 8179 2618 e+20= 3542 2484 8179 2618 00000

Example 1.4. Caryl Rusbult's Investment Model of Relationship
Scientific progress can be made without mathematical exactness or bit accuracy. Sometimes it is meaningful just to establish that factor A is positively (or negatively) related to factor B, when investigating a phenomenon involving factors A and B. Let us consider an example in psychology.

The domain problem has to do with domestic violence: why does a spouse being battered not leave an abusive relationship but stay committed to marriage? Professor Caryl Rusbult proposed a theory of investment model for close relationship:

$$\text{Commitment} \propto (\text{Satisfaction} \times \text{Investment})/\text{Alternative}$$

which can partially answer this question. A spouse's commitment to marriage is positively related to satisfaction and investment, but negatively related to alternatives. A battered spouse stays in an abusive relationship, not due to marriage satisfaction, but because she/he has invested heavily (e.g., having children) or has poor alternatives (e.g., without independent income).

Rusbult's investment model is not mathematically exact or bit accurate, but it is indeed a scientific progress, an inspirational theory which can lead to social policies and guides for individual actions. Computing and mathematics can be used to help test whether real data show that commitment is related positively to satisfaction and investment, but negatively to alternatives.

Doman scientists, such as psychologists, utilize their professional expertise and domain knowledge to advance their fields and contribute to society. Computer scientists complement their efforts by offering mental tools and computing hardware and software that feature automatically executed, bit-accurate, constructive abstractions of information transformation.

Constructive abstraction can be less intuitive. The confusion is partly due to the fact that the word "abstraction" refers to both an action and its outcome. That is, abstraction (the action) is the process of producing an abstraction (the outcome), which captures the essential aspect of an entity while ignores irrelevant aspects. All scientific disciplines have abstractions, but computer science emphasizes constructive, automatically executed abstractions of information transformation.

Constructive abstraction has three layers of meaning.

- The first is *abstraction*, namely, to abstract from concrete instances to the general concept. To quote from the Webster Dictionary: abstraction is "to form universal representations of the properties of distinct objects".
- The second layer of meaning is *constructive*, in that the resulting abstraction (the general concept) is constructive, which means that it is a step-by-step integration of more primitive symbols and operations.
- The third layer of meaning is *smart construction*. Computer science strives to understand the world and smartly construct abstractions based on such understandings. That is, computer science strives for smart constructions, not *ad hoc*, arbitrary actions or processes, although sometimes computational processes may use brute-force actions (e.g., exhaustive enumeration) and seemingly arbitrary random operations (e.g., randomly picking a number).

Refer to Table 1.1. The von Neumann model of computer says that a computer is comprised of a processor, a memory, and one or more input-output devices. It is an abstraction of many real computers, such as the student's laptop computer. It is constructive because a computer is composed of three more primitive parts in a unique way. The three primitive parts are processor, memory, and input-output devices. More details of the von Neumann model will be discussed in Chap. 2.

The concept of happiness is an abstraction of many individuals' happy feelings, but it is not constructive in that it is not a step-by-step integration of more primitive things. It is an abstraction of the first layer meaning but not of the second layer.

As an example of smart construction, we mention in Table 1.1 a Go program for computing Fibonacci numbers that utilizes a technique called dynamic programming. As discussed in Example 1.5 below, this program fib.dp-50.go is smarter and much faster than the program fib-50.go in Example 1.2.

In contrast, the binary executable file compiled from the same Go program, when damaged by destroying some bits, is not smartly constructive anymore. In fact, it ceases to be an abstraction, but is only a set of gibberish bits.

Example 1.5. Contrasting Four Processes to Find Fibonacci Number F(50)
The manual calculation process for Fibonacci numbers in Example 1.1 produces exact results, but it is not automatic and very tedious. Computing via fib-10.go is automatic, but it is slow when n gets large. Example 1.3 uses Binet's formula to compute Fibonacci numbers. It is faster but does not produce exact integer results.

Can we have a smarter way to compute Fibonacci numbers while insisting on getting exact integer results? Yes, we can. We will see in Chap. 2 that there is indeed a smarter way, called dynamic programming, to speed up the Fibonacci numbers computation significantly. The trick is to memorize intermediate results, avoiding repeated computations.

We will discuss the program details in later chapters. Here we only need to execute the four computational processes and contrast their behaviors, as summarized in Table 1.2. These four processes are the manual process in Example 1.1, and the three automatic processes using fib-50.go in Example 1.2, fib.binet-50.go in Example 1.3, and fib.dp-50.go in Fig. 2.9, respectively.

Table 1.2 Contrasting four computational processes for computing Fibonacci number F(50)

Process	Execution time	Produced result
Manual	135–600 s	May produce correct result 12586269025 May produce incorrect result, e.g., 1**3**586269025
fib-50.go	725 s	12586269025 Correct, exact result guaranteed
fib.binet-50.go	0.011 s	1.2586269024999998e+10 = 12586269024.999998 Correct, inexact result with a rounding error of 0.000002
fib.dp-50.go	0.059 s	12586269025 Correct, exact result guaranteed

The manual process is tedious thus prone to making mistakes. When the manual process produces a correct result, it is an exact integer value, i.e., 12586269025. The other three computational processes are automatic and guaranteed to produce correct results. The program using Binet's formula produces a correct floating-point number, 1.2586269024999998e+10, or 12586269024.999998, which is an approximate (inexact) value, with a **rounding error** (roundoff error) of 0.000002. The other two programs, fib-50.go and fib.dp-50.go, are guaranteed to produce correct and exact result F(50)=12586269025.

The manual process takes about 3–10 min to produce F(50)=12586269025, which actually consumes less time than the computational process utilizing the recursive program fib-50.go. The other two computational processes are much faster, by four orders of magnitudes.

1.2.2 The Eight Understandings Within

Looking from inside of the computer science field, namely, from the designer's perspective, we can understand computational thinking from eight aspects, as shown in Box 1.1. The eight understandings are pronounced as **Acu-Exams**.

The first understanding is automatic execution, which is actually the "A" in the ABC features without. In other words, step-by-step mechanic automatic execution of digital symbol manipulation is the most fundamental characteristic of computational thinking, both without and within. It underlies all the other seven understandings. Computer science studies logic that is automatic executable logic, algorithms that are automatic executed algorithms, abstractions that are automatic executed abstractions.

The other seven understandings address fundamental issues listed below, which are grouped into three parts of logic thinking, algorithmic thinking, and systems thinking, to be discussed in detail in Chaps. 3–5.

- **Logic thinking** addresses the issue: "What can be computed on a computer correctly?" To put it simply, *logic thinking makes computational processes correct.*
- **Algorithmic thinking** addresses the issue: "Given a computational problem, is there a smart way to solve it efficiently on a computer?" To put it simply, *algorithmic thinking makes computational processes smart.*
- **Systems thinking** addresses the issue: "How to construct a practical computing system, both general-purpose and specific?" To put it simply, *systems thinking makes computational processes practical.*

Box 1.1. Eight Understandings of Computational Thinking: Acu-Exams

- **A**: **Automatic execution.** Computational processes are automatically executed step-by-step on computers.
- **C**: **Correctness.** The correctness of computational processes can be rigorously defined and analyzed by computational models such as Boolean logic and Turing machines.
- **U**: **Universality.** Turing machine compatible computers can be used to solve any computable problems.
- **E**: **Effectiveness.** People are able to construct smart methods to solve problems effectively.
- **X**: **compleXity.** These smart methods, called algorithms, have time complexity and space complexity when executed on a computer.
- **A**: **Abstraction.** A small number of carefully crafted systems abstractions can support many computing systems and applications.
- **M**: **Modularity.** Computing systems are built by composing modules.
- **S**: **Seamless Transition.** Computational processes smoothly execute on computing systems, seamlessly transitioning from one step to the next step.

The issue in logic thinking can be further divided into two problems, which lead to Understandings C and U. First, what is correctness? It turns out that the correctness of computational processes can be rigorously defined and analyzed with the help of computational models such as Boolean logic and Turing machine. Second, is there a general-purpose computer that can correctly compute any computable entities? The answer is a rigorous Yes, in the form of Church-Turing Hypothesis. In addition, we can rigorously define what is not computable and provide concrete evidence, such as the Turing machine halting problem and Gödel's incompleteness theorem.

Algorithmic thinking involves how to make computational processes smart. Here we have two insights, i.e., Understandings E and X. Computer scientists have been able to rigorously define the concept of algorithms and have developed many types of smart algorithms. We will discuss several types, including divide-and-conquer and dynamic programming. Computer scientists are also able to rigorously define and analyze the time complexity and space complexity of many computational

Table 1.3 Execution time (seconds) of four programs for computing Fibonacci numbers F(n)

n	fib.go	fib.dp.go	fib.dp.big.go	fib.matrix.go
50	725	0.059	0.019	0.000012
500	Error	Error	0.026	0.000022
5,000,000	Error	Error	102	4.13
1,000,000,000	Error	Error	Killed after 2 days	187,160

problems and their algorithms. We will discuss the method of asymptotic analysis, utilizing the famous big-O notation and its cousins. We will also illustrate the famous problem of "P vs. NP" using examples, which is one of the seven Millennium Prize problems listed by Clay Mathematics Institute.

Systems thinking involves how to design practical systems for computational processes. A computing system may be a general-purpose computer like a student's laptop computer, or a specific computer application system such as WeChat. Computer science has progressed far from designing a system in arbitrary, ad hoc ways into more advanced ways. The essence of building a system is to use abstractions (Understanding A) to construct the system from modules (Understanding M), such that computational processes seamlessly transition from one step to the next step on the built system (Understanding S). We have millions of computer applications on billions of computer systems today. They are all supported by a small number of carefully crafted systems abstractions. We will discuss fundamental data abstractions and control abstractions in Chap. 5.

We use an example below to illustrate the objectives of logic thinking, algorithmic thinking, and systems thinking. The details will be discussed in Chaps. 3–5. Here we only need to run the programs and contrast their behaviors, to understand what is meant when we say logic thinking makes computational processes correct, algorithmic thinking makes them smart, and systems thinking makes them practical. It helps to hand out in class the involved programs fib.go, fib.dp.go, fib.dp.big.go, and fib.matrix.go.

Example 1.6. Fibonacci Computing in a Correct, Smart, and Practical Way
New students to computer science often have the implicit assumption that when the input data and the algorithm are correct, the program execution should successfully produce the correct result. The reality is more nuanced.

Let us compute F(n) by executing the four programs fib.go, fib.dp.go, fib.dp.big.go, and fib.matrix.go, for $n = 50$, 500, 5000000, and 1000000000, respectively. The behaviors of these programs are summarized in Table 1.3. Note that we have four versions of each program corresponding to the four input values of n. Thus, fib.dp-500.go is fib.dp.go when the value for n is set to 500.

The first program fib.go uses a straightforward recursive method. It is terribly slow (not smart), produces wrong results starting with $n = 93$ (not correct), and can only be used when n is small (not practical).

The second program fib.dp.go has the same incorrect and impractical issues as the fib.go program. However, it is smarter by using a dynamic programming algorithm. For $n = 500$, it takes less than a second to compute F(500).

```
> go run fib.dp-500.go
F(500) = 2171430676560690477
>
```

Compare this to the F(500) result we obtain by running fib.binet-500.go in Example 1.3, we see that running fib.dp-500.go generates a wrong output.

```
F(500) = 2171430676560690477      by fib.dp-500.go
F(500) = 1.394232245616976 7e+104  by fib.binet-500.go
```

It turns out that fib.dp.go computes correct Fibonacci numbers only up to F(92). For F(93), it generates a negative value: −6246583658587674878, an obviously incorrect result. The reason is that the 64-bit integer data type used in fib.dp.go is too small to hold F(93)= 12200160415121876738, which is larger than the largest 64-bit integer 2^{63}-1, or 9223372036854775807. The program has an **overflow** bug.

The fib.dp.big.go program fixes this overflow bug by using a data type called big. Int, allowing integers of arbitrarily **word length**, i.e., number of bits. We can comfortably compute not only F(500), but also F(5000000). The latter finishes in 102 s. Its result 7108285972. . .3849453125 has over 1 million digits.

When computing F(1000000000), i.e., n = 1 billion, fib.dp.big.go may run into a number of problems, such as "not enough memory". Even with enough memory, the program runs for two whole days without stopping, and has to be killed. We don't know how long it will execute to produce a result. The program is judged not practical for computing F(1000000000).

The last program fib.matrix.go optimizes further by using a "matrix exponentiation by doubling" algorithm. It takes 187160 s, or a little more than 3 h, to produce the result for F(1000000000), which is an integer with over 200 million decimal digits. The dominant part of the execution time is spent on conversion from the binary format result to the decimal format result, before printing. In any event, we finally have a program fib.matrix.go that is correct, smart, and practical, for computing Fibonacci numbers F(n) up to n = 1 billion.

The source code of all four programs can be found in Appendix 3.

1.2.3 A Research Viewpoint of Computer Science

To stimulate the students' curiosity and imagination, we discuss a viewpoint from the computer science research community. In 2004, the National Research Council of USA published a report, (called the US Academies Report in this book), which

Table 1.4 Concepts of this book compared to those in the US Academies Report

US Academies concepts	Concepts discussed in this book
Abstract computer	Turing machine, automata
Real computer	Laptop computer, WWW
Computer applications	Outcomes of the four projects, programming exercises
Symbol manipulation	Digital symbols from integer, character, image, to programs
Abstractions	Multiple abstractions, from circuit level to application level
Algorithms	Divide and conquer, dynamic programming
Artificial constructs	Students Computer for Quicksort
Exponential growth	P vs. NP, wonder of exponentiation
Fundamental limits	Turing computability, Godel's incompleteness theorems
Action associated with human intelligence	Reasoning by Boolean logic

summarized the fundamentals of computer science, including the essential character and salient characteristics, from researchers' viewpoint. These fundamental concepts are listed in Box 1.2.

It turns out that this textbook covers most essential character and salient characteristics from the US Academies Report, as shown in Table 1.4. Furthermore, more than half of knowledge units are presented in such way that students are able to pass Knuth's Test.

Knuth's Test: "The ultimate test of whether I understand something is if I can explain it to a computer. . . . In most of life, you can bluff, but not with computers."

Such knowledge units need to be learned in a mind-active, hands-on way. Simple memorization is not enough. We call such knowledge units **UKA units**, where UKA stands for Unity of Knowledge and Action. This pedagogic methodology of Unity of Knowledge-Action (知行合一) was borrowed from Wang Yangming (王阳明, 1472–1529), a Chinese educator from the Ming Dynasty. An essence of this methodology is to learn knowledge with mind-active, hands-on actions.

Box 1.2. Essential Character of Computer Science: A Research Viewpoint

Computer science is the study of *computers* and *what they can do*: the inherent power and limitations of *abstract computers*, the design and characteristics of *real computers*, and the innumerable *applications of computers* to solving problems.

Computer science research has the following salient characteristics:

(continued)

Box 1.2 (continued)

- Involves *symbols* and their *manipulation*.
- Involves the creation and manipulation of *abstractions*.
- Creates and studies *algorithms*.
- Creates *artificial constructs*, notably those unlimited by physical laws.
- Exploits and addresses *exponential growth*.
- Seeks the *fundamental limits* on what can be computed.
- Focuses on the complex, analytic, rational action associated with human *intelligence*.

1.3 Relation of Computer Science to Society

It helps understand computational thinking by looking at how computer science is related to the human society and our civilizations. Students probably have an intuitive feeling that computing is already everywhere. But how much? And why? We need to learn some basic facts and hypotheses:

- Computational thinking already permeates our civilizations. Its pervasiveness and fundamental importance are on par with capability of doing basic reading, writing, and arithmetic.
- There are fundamental reasons why computing is everywhere. Scholars have proposed several interesting hypotheses.
- Computer science is an attractive field, not just due to societal needs, but also because it is cool, exhibiting wonders and persuasions basic to our modern civilizations.

1.3.1 Computer Science Supports Information Society

Obviously, computing is already ubiquitous. A fact is that billions of people use smartphones to access Internet every day. By the statistics of ITU (the International Telecommunications Union), at the end of 2019, there were nearly 4 billion Internet users worldwide, penetrating over 51% of the global population.

An interesting question is: What's the age of the oldest computer user? An answer is 113 years old. In 2014, a lady in USA, who was born in 1900, had to lie about her age to sign on and use Facebook, as the Facebook sign-up page set the earliest birth year to 1905.

Although a lot of people have heard of "we are in the information age", fewer people appreciate how wide and deep the permeation is, still fewer people can explain why. We provide four essential facts and four hypotheses below.

Table 1.5 Digital economy data of 11 countries in 2016, in US$ Trillion

Country	Digital Economy Size	Percentage of GDP
United States	10.83	58.3%
China	3.40	30.3%
Japan	2.29	46.4%
Germany	2.06	59.3%
United Kingdom	1.54	58.6%
France	0.96	39.0%
South Korea	0.61	43.4%
India	0.40	17.8%
Brazil	0.38	20.9%
Russia	0.22	17.2%
Indonesia	0.10	11.0%

First fact: computer science directly supports the **information technology** (IT) industry. The **IT industry** provides and sells computer and network hardware products, software products, and services. The industry (producers) and the IT users (consumers) together form the IT market. The worldwide IT market had grown significantly, from only millions of US dollars in 1950s, billions in 1960s, to over one trillion dollars in year 2000, and 2 trillion dollars in year 2013. Today, market research firms such as IDC and Gartner use a larger metric, called the **information and communication technology** (**ICT**) spending, to measure the market size when the telecommunication sector is added to the market. In 2019, the worldwide ICT market is about US$3.7~5 trillion, by different tracking methods.

Second fact: computer science supports **digital economy**. Economists have observed a problem with the above measurement method of the IT or ICT industry: the revenue of many famous IT companies are not included in the above market data, because they sell little computer, network, telecommunication hardware, software, and services. These companies include Google, Facebook, Tencent, Baidu, Alibaba, etc. More than 90% of Google and Facebook's income are from advertising. As such they are advertising companies, not IT ones.

To better reflect the impact of ICT to the economy, economists established a new term, called *digital economy*. The definition and measurement methods of digital economy have not yet converged and stabilized. A 2017 study by Huawei and Oxford Economics utilized global data over three decades, and estimated that the global digital economy is worth 11.5 trillion US dollars. The top three digital economies are USA ($3.4 trillion), Europe ($2.9 trillion), and China ($1.5 trillion). The report observed that the world's digital economy grew two and a half times faster than global gross domestic product (GDP) over the past 15 years (2001–2016), almost doubling in size since the year 2000.

A group of Chinese digital economists, called China Info 100, published a study in 2018, which went one step further to broaden the scope of digital economy. Digital economy is divided into five sectors:

- Foundational digital economy (i.e., traditional ICT, 基础型信息经济),
- Productivity-enhancing digital economy (效率型信息经济),
- Convergence digital economy (融合型信息经济),
- Emergence digital economy (新生型信息经济), and
- Welfare digital economy (福利型信息经济).

The sum of all five sectors is the size of the digital economy. The 2016 numbers estimated by this report are shown in Table 1.5.

Third fact: computer science supports **information society**. Human civilizations have seen three main forms of society: the agriculture society, the industry society, and now the information society. Computer science does not just impact technology and economy, but also plays a central role in information society: a new megatrend and long-term phase of human civilization development. As evidence, the United Nations World Summit on the Information Society produced a document in 2003–2005, stating the following principle: “to build a people-centred, inclusive and development-oriented Information Society, where everyone can create, access, utilize and share information and knowledge, enabling individuals, communities and peoples to achieve their full potential in promoting their sustainable development and improving their quality of life, premised on the purposes and principles of the Charter of the United Nations and respecting fully and upholding the Universal Declaration of Human Rights.”

The fourth fact is a surprising one regarding human resources. Although billions of people use IT, the community of **IT professionals** is not large. Dr. David Grier, a former President of the IEEE Computer Society, defines IT professionals as people who have earned a bachelor degree and work in research, education, development, management and services of computing knowledge, products and services. He estimates that there are only about 3~10 million IT professionals worldwide. Let us take a middle number, say 7 million. That means there is roughly one IT professional per one thousand people of the world’s population. With the increasing demand of information society, it is no wonder that we see shortage of IT professionals in job market.

Now let us discuss the four hypotheses. They all try to answer the question: Why does computer science permeate our civilizations so widely and deeply? There are many explanations. We summarize four hypotheses in Box 1.3.

Chomsky’s digital infinity principle is due to Noam Chomsky, an American linguist. The idea is also expressed as “discrete infinity” and “the infinite use of finite means”, and can be traced back to Galileo. In essence, it says that that all human languages, no matter of which application domain or academic discipline, follow a simple logical principle: a limited set of digits are combined to produce an infinite range of potentially meaningful expressions. In other words, problems and knowledge in any domain can be expressed by the professional language of that domain. Any domain language can be expressed by digital symbols, thus amenable to computer processing.

Karp’s computational lens thesis is due to Richard Karp, an American computer scientist. We can understand Nature and human Society better through the computational lens. Why? Because Nature computes. Society computes. Many processes in Nature and human Society, traditionally studied in physical sciences, life sciences, or social sciences, are also computational processes. These processes are still physical processes, chemical processes, biological processes, psychological processes, business processes, social processes, etc. But viewing them as computational processes can bring in new perspectives and new value.

Babayan's gold metaphor is an observation made by Boris Babayan, a Russian computer scientist, in the HPC-Asia Conference in Beijing in the year 2000. Computing speed is like gold, a hard currency that can be exchanged for anything, be it new functionality, quality, cost, or user experience of products and services.

Boutang's bees metaphor is by Yann Moulier Boutang, a French economist. Why does ICT impact a much larger digital economy? We can liken ICT to bees. From an economic viewpoint, bees generate two outputs of value. The direct output is honey. The indirect output (economic externality) is pollination. Professor Boutang estimated that pollination has 28–373 times more economic value than honey. Likewise, the direct output of ICT is measured as the ICT market (about \$3.4~4.3 trillion in 2016). The indirect output is digital economy, which ICT enables and pollinates, and is multiple times larger (about \$11.5~24 trillion in 2016).

Box 1.3. Why Computer Science Permeates Our Civilizations
Chomsky's **digital infinity** principle: A finite set of digital symbols can be combined to produce infinite expressions in many domain languages.

Karp's **computational lens** thesis: Many processes in Nature and human Society are also computational processes. Nature computes. Society computes. We can understand Nature and Society better through the computational lens.

Babayan's **gold metaphor**: Computing speed is like gold, a hard currency that can be exchanged for anything.

Boutang's **bees metaphor**:ICT is like bees, producing two types of outputs. The indirect output (pollination) of bees has economic value that is orders of magnitude larger than the value of the direct output (honey). Similarly, the value of digital economy (indirect output) is much larger than that of the ICT market (direct output).

1.3.2 Computer Science Shows Three Wonders

The history of computer science has shown three wonders that are not often seen in other disciplines: the wonder of exponentiation, the wonder of simulation, and the wonder of cyberspace. These technology wonders are continuing, further stimulating innovations and applications.

Wonder of exponentiation: computing resources grow exponentially with time. Three such resources are listed in Box 1.4. We have Nordhaus's law for computer

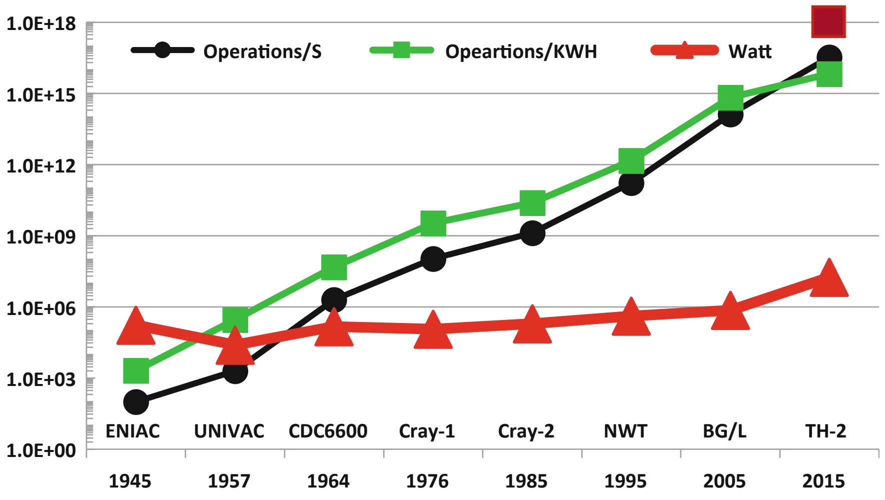


Fig. 1.7 Growth trends of computing speed, energy efficiency, and power consumption of the world's fastest computers (supercomputers) from 1945 to 2015. Special thanks to Drs. Gordon Bell, Jonathan Koomey, Dag Spicer and Ed Thelen for providing data for the first three computers

speed, Moore's law for the number of transistors on a semiconductor microchip, and Keck's law for communication bandwidth of an optical fiber.

It is remarkable that the wonder of exponentiation has existed for decades. It is even more remarkable that the wonder of exponentiation is likely to continue into future decades, despite the many seemingly unsurmountable technical obstacles. Common sense tells us exponential growth is not sustainable. But computing and communication speeds keep increasing exponentially.

Box 1.4. Laws Showing Wonder of Exponentiation

Nordhaus's law: computer speed grew exponentially with time, increasing 50% per year from 1945 to 2006. This observation was made in 2007 by Dr. William Nordhaus, an American economist.

Moore's law: the number of transistors in a semiconductor chip grows exponentially with time, doubling every 2 years or so. This observation was made in 1975 by Dr. Gordon Moore, an American engineer.

Keck's law: the data transmission rate of a single optical fiber grows exponentially with time, increasing about 100 times in 10 years. This observation was made in 2015 by Dr. Donald Keck, an American physicist and engineer.

Figure 1.7 shows growth trends of computer speed, energy efficiency (speed/energy), and power consumption data of the world's fastest computers from 1945 to 2015. We had a nice run for 60 years. Computer speed increased over a hundred trillion times. But recently we run into trouble: the energy efficiency did not improve as fast as speed anymore. Improving energy efficiency has become a top priority in computing system design.

A recent progress is domain-specific computing. If it is difficult for general-purpose computing to improve exponentially, can we increase speed and energy efficiency by focusing on a specific domain of computational processes? An example is the DianNao family of processors for deep learning workloads, which significantly improved the energy efficiency (purple box in Fig. 1.7).

The opposite to the wonder of exponentiation is the **curse of exponentiation**: many problems and algorithms have exponential complexity. But this challenge also serves as the source of many interesting researches and innovations. A case in point is protein folding, also called protein structure prediction. The problem is to computationally fold a protein into its three-dimensional structure. The brute-force approach needs 3^n operations, where n is size of the problem (usually $n =$ 300~600). Now $3^{300} \approx 10^{143}$ is a lot of operations. Computer scientists have been trying to find better algorithms that need much fewer operations, e.g., 1.6^n or even n^k, where k is a small constant. A recent progress is made by AlphaFold in 2020.

Wonder of Simulation: computer simulation provides a third paradigm for scientific enquiry, beyond theory and experiments. **Simulation** is to mimic physical or social processes by executing computer programs. The first computer simulation, also called computer experiment, was proposed and conducted in 1953 by physicists Enrico Fermi, John Pasta, and Stanislaw Ulam, to partially solve a physics problem later known as "the Fermi-Pasta-Ulam paradox". In doing so they invented "a third way of doing science…helped scientists to see the invisible and imagine the inconceivable", as commented by Professor Steven Sttogatz of Cornell University.

Example 1.7. Computer Simulation: Atoms in the Surf
The lecturer can show or ask the students to play with the video "Atoms in the Surf" in Supplementary Material. It shows how a supercomputer was used to simulate the collective motions of 9 billion aluminum and copper atoms, to reproduce a macro phenomenon known as the Kelvin-Helmholtz Instability.

The simulation begins with laminar flow such that the aluminum and the copper layers are heated to a temperature of 2000 K, and the relative velocity of the two layers is 2000 m/s. Computer simulation enables scientists to see not only the macro picture of how the aluminum-copper material evolves, but also the micro picture of each atom's state, every 2 femtoseconds.

Wonder of Cyberspace. The cyberspace enables designers to build new **virtual things** or **virtual worlds** that may not be possible in the physical world. Besides the human society and the physical space (Nature and human-built things), computer science helps create a new space called the cyberspace, consisting of computational processes running on computers (recall Fig. 1.1). This is a salient feature of computer science: creates artificial constructs, notably unlimited by physical laws (recall Box 1.2). Real examples abound, such as the following.

Can we build a shopping mall hosting a million vendors? This is difficult to do in the physical world, but is already a reality in cyberspace. An example is the electronic commerce services provided by the company Alibaba Group, which host over ten million vendors through its Tmall and Taobao platforms in year 2020.

Can we build a bookstore holding a billion books? Again, we can, but in cyberspace. Think of Amazon.com. We can also build a library in cyberspace, where the collections are so large that we would need a thousand-floor library to hold the books in the physical world.

Sometimes, cyberspace works together with human society and physical world to create a Human-Cyber-Physical ternary computing system. In April 2019, scientists published a research paper containing the first photographs of a blackhole. The blackhole is 55 million light years away from the Earth. To take a photograph of it, we need a telescope as big as our planet. Scientists utilized multiple physical telescopes in several continents, to form an Earth-diameter virtual telescope. Imaging data were captured in April 5–11 2017 by these physical telescopes and stored in hard disks, which were then shipped to supercomputers for data correlation and reduction. These computation and post processing took 2 years, before the images of this blackhole were published in April 2019.

1.3.3 Computer Science Has Three Persuasions

Computing has a long history. But modern computer science is quite young. There is no universally agreed birthdate of modern computer science. Some scholars put the birth year to be 1936, when Alan Turing published his seminal paper on computability. Some choose 1945, when the first electronic digital computer, ENIAC, was built. Some would say 1962, when the first Department of Computer Science was established at Purdue University.

Although so young, computer science has grown into a rich field with many interesting problems, scientific discoveries, and engineering techniques, which combined produce wide and deep societal impact. A downside of this richness is that there are too many buzzwords and even hypes associated with IT or ICT, bewildering to new students of computer science.

The reality is that at its core, computer science has a number of basic persuasions that are relatively stable. What changes is the scope, refinement, manifestation, and embodiment of these basic persuasions or visions. Jim Gray in 1999 noted three such fundamental visions. We slightly revise his viewpoints and call the three visions as problems: Babbage's problem, Bush's problem and Turing's problem, to emphasize that they are fundamental problems worthy of continued study. These problems are summarized in Box 1.5.

Babbage's problem: How to build efficient, programmable computers?

Here Babbage is Charles Babbage, a British computer scientist and professor at Cambridge University. His original vision was to build a programmable computer with information storage that could compute much faster than humans. In 1883,

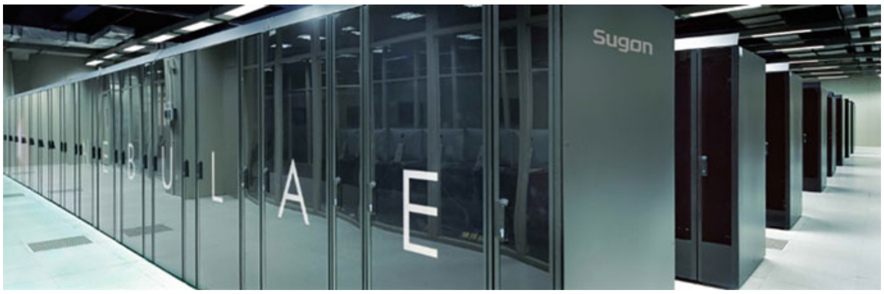


Fig. 1.8 A server example: Sugon Nebulae supercomputer hosted in a machine room

Babbage proposed the design of a mechanical digital computer called Analytic Engine. Although not built, this is considered the first design of a general-purpose digital computer capable of automatic execution. Ada Lovelace, who wrote a program for this computer to compute Bernoulli numbers, is generally recognized as the first computer programmer.

Box 1.5. Computer Science Has Three Persuasions
Babbage's problem: How to build computers? More specifically, how to build efficient, programmable computers? Efficiency may mean degree of automation, computational speed, or energy efficiency (computational speed per Watt).

Bush's problem: How to use computers? More specifically, how to use computers conveniently and effectively in solving problems? This calls for new conceptions on how humans, computers and information interact.

Turing's problem: How to make computers intelligent? More specifically, how to make computing systems intelligent? Here intelligence generally refers to approaching intelligent behaviors akin to humans.

Today, Babbage's original vision is already realized. We have in fact expanded his vision significantly. Three types of computers exist:

- **Client**-side computers. These are computers most familiar to us, as humans (clients) directly use them. Examples include personal computers (PCs) such as desktop computers and laptop computers, and mobile devices such as smartphones and various smart pads.
- **Server**-side computers. They are also simply called servers, often hosted in glassed-off machine rooms or Internet datacenters. Users do not directly see these computers, but indirectly use them through client devices. Examples include on-premise servers in a company, cloud computing servers hosted in Internet datacenters, and supercomputers. An example is shown in Fig. 1.8.

- **Embedded** computers. These are computers embedded (hidden) in other systems. People do not see a computer, but see a non-computer system, such as a microwave oven, a refrigerator, a car, or a pair of shoes.

Dr. Gordon Bell offers a finer classification of computers from the historical perspective. His insight is based on observation of several decades, and becomes known as **Bell's law**: Computers develop by following three design styles, to generate a new computer class roughly every 10 years. The three design styles are: (1) develop the most capable computers with price as a secondary consideration; (2) improve the performance but maintain a constant price; (3) reduce the price as much as possible to produce a new "minimal-priced computer". About a dozen computer classes formed in the six decades from 1950 to 2007. Ten are listed below.

- Server-side computers hosted in on-premise machine rooms or datacenters

  1. Supercomputers, the most capable computers
  2. Mainframes, such as IBM S360
  3. Minicomputers, such as DEC PDP-11
  4. Clusters (systems of interconnected computers), such as IBM SP2

- Client-side computers directed used by humans

  5. Workstation, with graphics processing and display capability
  6. Personal computers (desktop PC), such as Apple 2
  7. Portable computers, such as laptop computers
  8. Dedicated personal devices, such as a game device, a digital camera
  9. Smartphone, such as Apple iPhones
  10. Wearable devices, such as a smart watch

There are already billions of computers of various classes worldwide. Many in the IT community believe that this is still only the beginning. By 2040, there may be trillions of computers worldwide. Most of them will be smart things that interact with the physical world, also known as Internet of Things (IoT) devices. Research opportunities abound for new classes of computers, both server-side and client-side.

Bush's problem: How to use computers effectively?

Bush here refers Vannevar Bush, an American engineer and an MIT professor. He proposed a vision called "Memex" in an influential article "As We May Think" published in 1945, and revisited 20 years later in another article "Memex Revisited" in 1965. Memex is rich concept including at least two characteristics: (1) every scientist should have a personal computer that stores all human knowledge; and (2) the scientist can easily access information and knowledge he needs, by associating one scientific record to another record. This association concept is called **hypertext** today and appears in technology such as the World Wide Web.

In essence, Bush urges us to study and revisit the relationship between thinking man and the sum of human knowledge, beyond the mechanic relationship between a user and his computer device. From a more practical perspective, Bush's problem directs our attention to the usage mode of computing systems. A **usage mode** consists of the following considerations:

- The intended user community, e.g., scientists in Bush's Memex example.
- The organization style of information (and knowledge), e.g., hyperlinked records in Bush's Memex example.
- The style of human-computer interaction, e.g., interactive read, write, and select by following hyperlinks in Bush's Memex example.

Usage modes visibly impact society. Widespread adoption of a usage mode often signifies a new computing market. We have made great strides on realizing Bush's vision. From the human-computer interaction viewpoint, we have seen the following usage modes in the 70-year history of modern computer science.

- **Batch** processing mode. A user submits a computational job (including program and data) to the computer, and then waits for seconds, hours, days, or months before the computer returns the result.
- **Interactive** computing mode. A user interacts with the computer instantly. For instance, when entering 3000 words to form a file on a PC, the user sees instant screen output of each entered character, without having to wait for all 3000 words having been entered and processed in a batch processing way.
- **Personal** computing mode. Early computers, accessed via either the batch or the interactive modes, are shared among multiple users. A personal computer (PC) is dedicated to a single user's usage.
- **GUI** mode. Early computers are accessed via a character interface. Later computers provide graphic user interface (GUI).
- **Multimedia** mode. The GUI mode is extended to include not only graphics, but also multiple media types such as images, audio and video.
- **Portable** computing mode. Now we can carry a computer around, in a bag, in our pocket, etc. An example is a laptop computer.
- **Network** computing mode. Now we can access computing resources via computer networks, e.g., local area network, the Internet, or the World Wide Web. An important network computing mode is called **cloud computing**, where many resources are located in the server side (in the cloud datacenters), and accessed through the network via client-side devices.
- **Mobile Internet** mode. This mode combines the portable and the network modes. An obvious example is to use WeChat on a smartphone.

Fundamentally, Bush's problem is about how to best connect people, computers, and information. This persuasion is continuing and new research opportunities constantly appear, especially with respect to the trend of Human-Cyber-Physical ternary computing systems. A concrete example is research in touchless interaction, which upends traditional ways to use computers by touch (via keyboard, video display and mouse) in a PC, or by touchscreen in a smartphone.

Turing's problem: How to make computing systems intelligent.

Here Turing is Alan Turing, a British computer scientist and a founding father of modern computer science. Turing's problem can be rephrased as "how to make computer application systems intelligent", emphasizing the intelligent applications of computers. Broadly speaking, there are three types of computer applications.

- The first type is **scientific computing** applications, mainly for scientists and engineers. Their main workloads are to solve equations, to do computer simulations, and to process scientific data.
- The second type is **enterprise computing** applications, mainly for organizations such as companies, government agencies, and not-for-profit institutions. The workloads include business workflows, transaction processing, data analytics, decision support, etc.
- The third type is **consumer computing** applications, for individual consumer users (the masses). Enterprise computing is also called **business computing**. This is why the students may have heard phases such as "to B" (products or services for business) and "to C" (products or services for consumers), or even B2B, B2C, C2C, and C2B.

Among his fruitful research results, Alan Turing made two fundamental contributions to computer science. In 1936, Turing rigorously defined the concept of computability. In 1950, Turing proposed a test for machine intelligence and argued that computer applications could eventually become intelligent.

From a practical application's viewpoint, Turing's first paper shows that any computable problem, be it a scientific computing problem, a business computing problem, or a consumer computing problem, can be solved by computer applications. Here computable problems are precisely defined as computable numbers produced by a precisely defined computer, later called the Turing machine. Any real number, such as any Fibonacci number or the circular constant π, is computable if its decimal digits can be written down by a Turing machine automatically in a sequence of step-by-step elementary operations. Turing also shows that there are problems not computable. An example is the Entscheidungsproblem (German for "decision problem"), which is a fundamental mathematics problem formulated in 1928 by David Hilbert and Wilhelm Ackermann. It asks: is there an algorithm to decide whether a statement is a theorem in a given set of axioms? Turing's paper gave a negative answer.

Turing's second paper went further: not only computable problems are solvable by computer applications, but also some of these applications can be as intelligent as humans. Turing did not offer a proof, but presented an interesting argument. He proposed a test, later called the **Turing Test**, to show that a computer is intelligent if a human observer cannot distinguish the computer from a human player in an Imitation Game. There are three parties (two humans and a computer) in this game. A human interrogator C asks questions of two players A and B in another room, to determine whether A or B is a computer. The computer passes the Turing test if "[the] average interrogator would not have more than 70 per cent chance of making the right identification after five minutes of questioning".

Seventy years have passed since Turing's 1950 paper, and we have made significant progress regarding Turing's problem. Many computer application systems show some intelligent behavior akin to humans. Computers beat human players in many games, such as Chess, Go, Poker, and DOTA. Computer applications in pattern recognition, language translation, autonomous vehicles, robotics, and machine learning are already in practical use. This subfield of computer science is called artificial intelligence (AI) and has attracted much attention.

1.3.4 Computational Thinking Is a Symphony

We have briefly discussed computer science and computational thinking. A set of concepts have already emerged: encoding of domain problems into cyberspace, computational process as digital symbol manipulation by a sequence of step-by-step elementary operations, the ABC features without, the eight understandings within (Acu-Exams), three wonders, and three persuasions. These multitudes of concepts reflect the richness of the field, but may be bewildering to new students. A key to handle this richness and complexity is to view computer science as one thing: a symphony. It is not simply a pile of those particularities, but a synergy of them.

The richness is a fact of the field. Different scholars voiced different conceptions of computer science and computational thinking. Three examples follow.

- Professor Georg Gottlob, of Oxford University, believes that computer science is the continuation of **logic** by other means, analogous to Clausewitz's saying that war is the continuation of politics by other means.
- Professor Richard Karp, of the University of California at Berkeley, promotes the concept of computational lens (also known as algorithmic lens), emphasizing solving scientific and societal problems through the lens of **algorithms**.
- Dr. Joseph Sifakis, of the French National Center for Scientific Research, advocates **system** design science as a basic goal of the computer science field.

These three different viewpoints offer different perspectives on the same thing. Computational thinking is a synergy of all of the concepts above. We call this principle Yang Xiong's **Principle of Harmony** (扬雄和谐原理), as the Chinese scholar Yang Xiong (53 BCE–18 CE) presented a similar principle in around year 2 BCE, in his work *The Canon of Supreme Mystery* (太玄经), a classic of 81 verses on creativity. Professor Michael Nylan produced an English translation. Yang Xiong invented a ternary symbol system (⚊, ⚋, 𝌀). A verse is called a *head* (首).

Box 1.6 shows part of the verse named *Cha* (差,, ternary symbol 𝌐), translated roughly as *diversity* or divergence.

Box 1.6. Computer Science Is a Symphony

《太玄经·差首》： 𝌐 帝由群雍，物差其容。

Head *Cha* (Diversity) 𝌐: The way emerges from the multitude of harmonies, where things diverge in their appearances.

Computer science is like a musical symphony. Many instruments produce different sounds, but all instruments play the same music. Each instrument offers its distinct contribution. The diversity of their differences creates a harmonic whole of the symphony. Logic thinking, algorithmic thinking and systems thinking together produce the totality of computational process, that is correct, smart, and practical.

1.4 Exercises

For each exercise, select all correct answers. A selection including all and only correct answers receives full score. A selection including one or more wrong answers receives 0 score, but no penalty.

1. Refer to Fig. 1.1.
   (a) The domain problem in the target domain must be a mathematic problem. Problems in other domains must be first encoded into mathematical problems, before computer science can play a role in problem solving.
   (b) The cyberspace consists of computational processes executing on computer systems. By this definition, the ancient Egyptian civilization did not have cyberspace, since there were no computers at that time.
   (c) By the above definition of cyberspace, the ancient Egyptian civilization DID have cyberspace, since ancient Egyptians calculated tax based on flood level data of the Nile river measured by nilometers. The computational process (tax calculation) was executed by computers in the forms of tax officials, nilometers and possibly other devices.
   (d) The cyberspace is the union of the physical space and the human society.
2. A binary digit (one bit) can be used to represent the following entity:
   (a) The traffic light colors of Red, Yellow, Green.
   (b) The answer to a Yes/No question.
   (c) The state of an On/Off switch.
   (d) The current time displayed on a digital clock.
3. Refer to Example 1.1 and Fig. 1.3.
   (a) The algorithm in Fig. 1.3a is a digital symbol, since it denotes the algorithm to compute F(10), is represented by a number of English characters, and each English character can be represented by a number of bits.

(b) The program fib-10.go is a digital symbol, since it denotes a high-level language program and is representable by a number of bits.
(c) The program fib-10 is a digital symbol, since it denotes a machine code program and is representable by a number of bits.
(d) The screen output F(10)= 55 in Fig. 1.3c is a digital symbol, since it denotes the entity of a program's output and is representable by a number of bits.
(e) The action of a human programmer entering the command "go build fib-10.go" is not a digital symbol, since it is not representable by a number of bits. The string "go build fib-10.go" is a digital symbol, but it is the result of the action, not the action itself.

4. Three types of code are shown in Example 1.1: high-level language program, binary program, and command (or shell command). How is each of the three types of code processed by the computer system? Put the correct capital letter in the parentheses of each line below.

(a) The high-level language program "fib-10.go" is ().	X: executed
(b) The binary program "fib-10.go" is ().	Y: interpreted
(c) The command "go build fib-10.go" is ().	Z: compiled

5. Refer to Example 1.1. Suppose "F(10)" is changed to "F(50)" in program fib-10.go. The screen output in Fig. 1.3c should become:

(a) F(10)= 55
(b) F(10)= 12586269025
(c) F(50)= 55
(d) F(50)= 12586269025

6. Refer to Example 1.1. Suppose "fibonacci(10)" is changed to "fibonacci(50)" in program fib-10.go. The screen output in Fig. 1.3c should become:

(a) F(10)= 55
(b) F(10)= 12586269025
(c) F(50)= 55
(d) F(50)= 12586269025

7. Refer to Example 1.1. Suppose "// Output F(10)" is changed to "// Output F(50)" in program fib-10.go. The screen output in Fig. 1.3c should become:

(a) F(10)= 55
(b) F(10)= 12586269025
(c) F(50)= 55
(d) F(50)= 12586269025

8. Refer to Example 1.1. Suppose "10" is changed to "50" in program fib-10.go. The screen output in Fig. 1.3c should become:

(a) F(10)= 55
(b) F(10)= 12586269025

   (c) F(50)= 55
   (d) F(50)= 12586269025

9. Refer to Example 1.1. Why do we need the compiler to compile program fib-10.go into program fib-10?
   (a) The compiler checks for compile-time errors in the high-level language program, such as various syntactic errors.
   (b) The compiler checks for runtime errors.
   (c) Program fib-10.go is a machine code program.
   (d) The computer only understands and executes a machine code program.

10. Refer to Example 1.1. The command "go build fib-10.go" looks like a high-level language statement and seems to directly execute on a computer. Why does this not contradict to the assertion that "computer only understands machine code"?
    (a) A command is not a program, therefore can directly execute on a computer.
    (b) A command is a high-level language program and is interpreted into machine code by a command interpreter called shell. The command seems to execute directly, because the interpretation is done automatically and behind the scene.
    (c) The command is a short statement, and the computer can understand single and short high-level language statements.

11. Why is it much easier for human to understand a high-level language program than a machine code program?
    (a) High-level language programs are written by highly skilled programmers.
    (b) High-level language programs execute much faster than machine code.
    (c) High-level language programs are shorter than machine code.
    (d) A high-level language is similar to a natural language.

12. Regarding overflow, which of the following statements is correct?
    (a) An overflow error occurs when the result value is too large for the bits available. For instance, the value 9 is too large for a 4-bit integer (overflow), but not too large for a 4-bit unsigned integer (no overflow).
    (b) An overflow error occurs when the absolute value of the result is too large for the bits available. For instance, the absolute value of -9 is $9=1001_2$, which can be held in 4 bits. Thus, -9 does not cause overflow for a 4-bit integer representation.
    (c) Rounding errors (roundoff errors) are a type of overflow errors.
    (d) Overflow errors are a type of roundoff errors.

13. Eight bits are used to represent an integer value. Which will result in overflow?
    (a) When the integer is -256.
    (b) When the integer is -129.
    (c) When the integer is -64.
    (d) When the integer is 64.

   (e) When the integer is 129.
   (f) When the integer is 256.

14. Two computers compute 2.0/7.0 and obtain two different results. Why?
   (a) An overflow error occurs.
   (b) A compilation error occurs.
   (c) A roundoff error occurs.
   (d) One computer is a human, and he made a mistake calculating 2.0/7.0.

15. When looking from outside, computational thinking has three features without, called the ABC features. They are:
   (a) Automatic execution
   (b) Binary representation
   (c) Computational abstraction
   (d) Constructive abstraction

16. Bit-accuracy in a computational process means:
   (a) Every operation of the computational process generates a result that is accurate and precise up to every bit.
   (b) The computational process generates a correct integer result.
   (c) The computational process generates a final result value that is precise up one binary digit after the decimal point.
   (d) The computational process generates a final result with statistical significance, i.e., the p-value less is than 0.05.

17. When looking inside, computational thinking has eight understandings within, with an acronym Acu-Exams. They are:
   (a) Automatic execution
   (b) Correctness and Universality in logic thinking
   (c) Effectiveness and Complexity in algorithmic thinking
   (d) Abstraction, Modularity and Seamless Transition in systems thinking

18. The Information Technology (IT) industry provides:
   (a) Computer hardware products, such as laptop computers and servers
   (b) Network hardware products, such as WIFI routers and network cards
   (c) Computer software products, such as operating systems and Web browsers
   (d) Internet services, such as search engine and video sharing

19. ICT refers to the Information and Communication Technology industry. It provides:
   (a) Computer and network hardware products, such as desktop computers and smartphone devices
   (b) Computer software products, such as operating systems and scientific computing software

   (c) Internet services, such as search engine and video sharing
   (d) Telecommunication services, such as telephone services and Internet connection services

20. The worldwide ICT spending in 2019 was about:

   (a) 40 billion US dollars
   (b) 400 billion US dollars
   (c) 4000 billion US dollars, or 4 trillion dollars
   (d) 40 trillion US dollars

21. The worldwide population is about 7.8 billion people in year 2019. How many of them were estimated as IT professionals?

   (a) 780 thousand, that is, one IT professional serving 10000 people
   (b) 1 million, that is, one IT professional serving 7800 people
   (c) 7.8 million, that is, one IT professional serving 1000 people
   (d) 78 million, that is, one IT professional serving 100 people

22. About how much percentage of the worldwide population are computing professionals (also known as IT professionals)?

   (a) 0.01%
   (b) 0.1%
   (c) 1%
   (d) 10%

23. The following statements regard the four hypotheses explaining the impact of computer science.

   (a) When Richard Karp said Nature computes and Society computes, he meant that many processes in natural sciences and social sciences can be viewed as computational processes.
   (b) When Richard Karp presented the computational lens thesis, he meant that he can turn his smartphone's camera into a telescope to see stars.
   (c) When Boris Babayan proposed his gold metaphor, he meant that one can sell one's computer for gold.
   (d) When Yann Moulier Boutang proposed his bees metaphor, he meant that ICT produces direct economic value (like bees producing honey), as well as indirect value (like bees pollinating), and the indirect value is much larger than the direct value.

24. The following explains why computer science has wide impact.

   (a) Computer science is useful for many fields, because there are infinite many computer programs. This is known as the Chomsky digital infinity principle.
   (b) Computer science is useful for many fields, because many processes in those fields can be viewed as computational processes, i.e., processes of information transformation. This is known as Karp's computational lens thesis.

(c) Wires in microchips of computers should be made of gold, to resist corrosion and provide reliability. This is known as Babayan's gold metaphor.
(d) ICT produces indirect economic value much larger than its direct value. This is analogous to bees producing honey and doing pollination. The indirect value (pollination) is much larger than the direct value (honey). This is known as Boutang's bees metaphor.

25. According to Boutang's bees metaphor, the worldwide digital economy has a much large value than the worldwide ICT spending number. The worldwide digital economy in 2016 was valued at about:

(a) 150 billion US dollars.
(b) 1.5 trillion US dollars.
(c) 15 trillion US dollars.
(d) 150 trillion US dollars

26. The following statements are about wonder of exponentiation.

(a) Computer speed has increased exponentially with time since 1945.
(b) Computer speed has increased exponentially with time since 1800.
(c) Computer speed will increase exponentially with time till 2045.
(d) Computer speed will increase exponentially with time till 2800.

27. The following statements are about wonder of simulation.

(a) Computer simulation of car crashes is more economic and less dangerous than physical tests of car crashes.
(b) Simulated car crash tests have fully replaced physically crashing cars.
(c) Simulated car crash tests can provide insights on the design of the cars.
(d) Simulated car crash tests can help formulate and verify the hypothesis that drivers with dementia are more likely to experience accidents.

28. The following statements are about wonder of cyberspace.

(a) All things and processes in the cyberspace also appear in the physical world, because Nature computes and Society computes.
(b) All things and processes in the cyberspace also appear in the physical world, because computers can only simulate physical processes governed by scientific laws.
(c) Things and processes in the cyberspace can be absent in the physical world, because a tenet of computer science is to creates artificial constructs, notably those unlimited by physical laws.
(d) The cyberspace can help create *virtual* things different from traditional physical things. An example is the Event Horizon Telescope, which is an Earth-diameter *virtual telescope* that was used to successfully take photographs of a blackhole.

29. The following statements are about Babbage's Problem.
    (a) A laptop computer is a server-side computer.
    (b) A laptop computer is a client-side computer.
    (c) A laptop computer is an embedded device.
    (d) A laptop computer is a computer cluster.
30. The following statements are about Bush's Problem.
    (a) When a user is browsing the Web using a home PC, the user-computer is working in the batch mode for scientific computing applications.
    (b) When a user is browsing the Web using a home PC, the user-computer is working in the interactive mode for consumer computing applications.
    (c) C2C stands for Computer-to-Computer applications.
    (d) C2C stands for Consumer-to-Consumer applications.
31. The following statements are about the Turing Test.
    (a) The Turing Test is used to test how well a computer can drive an autonomous vehicle.
    (b) The Turing Test is used to test how well a computer can recognize the object in a picture, e.g., identifying the object as a cat or a dog.
    (c) The Turing Test is used to test whether a computer can beat human in Chess.
    (d) The Turing Test is used in a dialogue between a human interrogator and two interrogated parties (a human and a computer) to see if the interrogator can correctly tell the computer apart from the human.
32. What does it mean that "computer science is a symphony"?
    (a) It means that multiple computers on the Internet can work together in real time to play Beethoven's Ninth Symphony.
    (b) It means that multiple laptop computers in the same classroom can work together in real time to play Beethoven's Ninth Symphony.
    (c) It means that computer science is the synergy of logic thinking, algorithmic thinking and systems thinking.
    (d) Designing a computer application system only involves systems thinking, to make the application system practical. It does not need to involve logic thinking or algorithmic thinking, which are too theoretical.

1.5 Bibliographic Notes

The chapter quotation is from an interview of Donald Knuth by Quanta Magazine in February of 2020 [1]. Rusbult's investment model of relationship can be found in [2]. Computer science fundamentals are discussed in [3, 4]. Digital economy data and the principle of information society are presented in [5–7]. The concepts of Chomsky's digital infinity, Karp's computational lens, and Boutang's bees metaphor can be found in [8–10]. Historical trends of computing-related metrics are shown in

[11–15]. A recent progress in high-accuracy protein structure prediction is reported in [16]. Computer simulation is discussed in [17, 18]. Examples of Human-Cyber-Physical ternary computing systems are discussed in [19, 20]. Discussions on Babbage's problem, Bush's problem, and Turing's problem can be found in [21–26]. Nylan [27] provides an English translation with commentary of 太玄经, The Canon of Supreme Mystery.

References

1. D'Agostino S (2020) The computer scientist who can't stop telling stories. Quanta Mag. https://www.quantamagazine.org/computer-scientist-donald-knuth-cant-stop-telling-stories-20200416
2. Rusbult C, Martz J (1995) Remaining in an abusive relationship: an investment model analysis of nonvoluntary dependence. Pers Soc Psychol Bull 21(6):558–571
3. Wing JM (2006) Computational thinking. Commun ACM 49(3):33–35
4. US National Research Council (2004) Computer science: reflections on the field. In: Reflections from the field. National Academies Press, Washington, DC
5. Huawei and Oxford Economics (2017) Digital spillover: measuring the true impact of the digital economy. https://www.huawei.com/minisite/gci/en/digital-spillover/index.html
6. China Info 100 (2018) The 2017 China digital economy development report. http://www.chinainfo100english.com/201803/432.html
7. World Summit on the Information Society (2003) Building the information society: a global challenge in the new millennium. Declaration of Principles
8. https://www.wikizero.com/en/Digital_infinity
9. Karp RM (2011) Understanding science through the computational lens. J Comput Sci Technol 26(4):569–577
10. Moulier-Boutang Y (2007) Cognitive capitalism and entrepreneurship: decline in industrial entrepreneurship and the rising of collective intelligence. In: Conference on capitalism and entrepreneurship. Cornell University, Ithaca, 28–29 Sept 2007
11. Nordhaus WD (2007) Two centuries of productivity growth in computing. J Econ Hist 67(1):128–159
12. Moore GE (1975) Progress in digital integrated electronics. In: Electron devices meeting, vol 21, pp 11–13
13. Hecht J (2016) Great leaps of light. IEEE Spectr 53(2):28–53
14. Xu ZW, Chi XB, Xiao N (2016) High-performance computing environment: a review of twenty years experiments in China. Natl Sci Rev 3(1):36–48
15. Chen Y, Chen T, Xu Z, Sun N, Temam O (2016) DianNao family: energy-efficient hardware accelerators for machine learning. Commun ACM 59(11):105–112
16. Jumper J, Evans R, Pritzel A, Green T, Figurnov M, Tunyasuvunakool K, et al (2020) High accuracy protein structure prediction using deep learning. In: Fourteenth critical assessment of techniques for protein structure prediction (abstract book), pp 22, 24
17. Strogatz S (2003) The real scientific hero of 1953. New York Times, 4 March 2003
18. Richards DF, Krauss LD, Cabot WH et al (2008) Atoms in the surf: molecular dynamics simulation of the Kelvin-Helmholtz instability using 9 billion atoms. https://arxiv.org/abs/0810.3037, www.youtube.com/watch?v=Wr7WbKODM2Q
19. Xu ZW, Li GJ (2011) Computing for the masses. Commun ACM 54(10):129–137
20. Akiyama K, Alberdi A, Alef W et al (2019) First M87 event horizon telescope results. IV. Imaging the central supermassive black hole. Astrophys J Lett 875(1):L4
21. Gray J (2003) What next?: A dozen information-technology research goals. J ACM 50(1):41–57

22. Bell G (2008) Bell's law for the birth and death of computer classes. Commun ACM 51(1):86–94
23. Bush V (1945) As we may think. Atlantic Monthly 176(1):101–108
24. Bush V (1991) Memex revisited. In: Nyce J, Kahn P (eds) From Memex to typertext: Vannevar Bush and the mind's machine. Academic Press, Boston, pp 197–216
25. Turing AM (1936–1937) On computable numbers, with an application to the Entscheidungsproblem. Proc Lond Math Soc 42(2):230–265
26. Turing AM (1950) Computing machinery and intelligence. Mind 59(236):433–460
27. Nylan M (1993) The Canon of supreme mystery by Yang Hsiung: a translation with commentary of the T'ai Hsuan Ching. SUNY Press, Albany

Chapter 2
Processes of Digital Symbol Manipulation

A physical symbol system has the necessary and sufficient means for general intelligent action.
—Allen Newell and Herbert A. Simon, 1976

Symbols are carriers of human civilizations. Digital symbols are carriers of the modern human civilizations. Digital symbol manipulation is at the core of computer science. We discuss several examples of digital symbol manipulation in this chapter, to show that data are digital symbols, programs are digital symbols, and computer systems are a platform for digital symbol manipulation.

These examples are (1) binary-decimal number conversion, (2) representing integers, (3) representing characters, (4) writing simple programs, (5) writing programs relating character strings to integers, (6) writing programs to compute large Fibonacci numbers in two methods, recursive and dynamic programming.

These examples assume a von Neumann model of computer, which will also be introduced with a detailed example of step-by-step execution of instructions, to show how a computer works.

2.1 Data as Symbols

Many quantities in the physical world have **analog values**. Such a quantity has continuous values. They are basically real numbers, but often represented by a finite number of digits according to the application requirement on precision. For instance, Fig. 2.1 shows the analog quantity of monthly average high temperature of Beijing in 2019, which have continuous values. This analog quantity of temperature can be converted into a digital quantity by **discretization**, i.e., using discrete values shown in the following table in both binary and decimal formats. There is a question mark for the seventh month (July), which will be elaborated in an exercise.

Z. Xu, J. Zhang, *Computational Thinking: A Perspective on Computer Science*,
https://doi.org/10.1007/978-981-16-3848-0_2

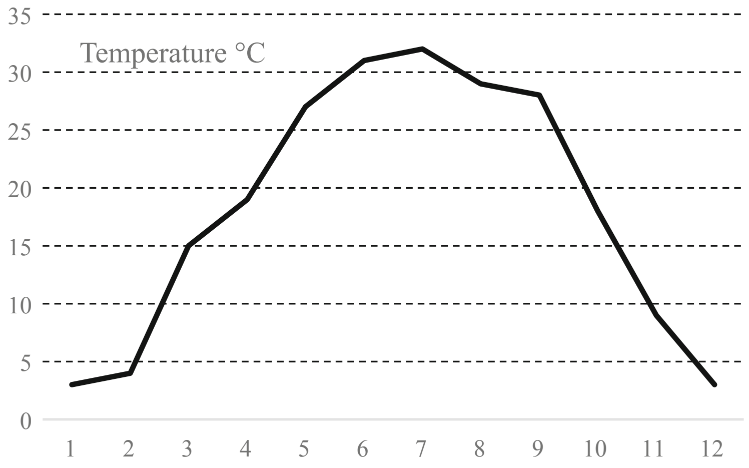


Fig. 2.1 An analog quantity: average high Temperature value in Beijing in year 2019

Month	1	2	3	4	5	6	7	8	9	10	11	12
Temperature °C	00011	00100	01111	10011	11011	11111	?	11101	11100	10010	01001	00011
	3	4	15	19	27	31	32	29	28	18	9	3

Discretization maps continuous analog values to non-continuous discrete values. There is no intermediate value between two consecutive discrete values. Discrete values are also called digital values or digital symbols. Three terms are often used in computer science regarding digital values: bit, byte, and word.

- **Bit** is the smallest digital symbol that can have a value of 0 or 1.
- **Byte** is a group of 8 bits. It is the smallest unit used by a typical computer when storing digital symbols in memory. When a load or store instruction is executed to access the memory, the computer accesses at least one byte. This is why memory in most computers are called **byte-addressable** memory.
- **Word** is a group of bits. It is the smallest unit used by a typical computer when processing digital symbols (or digital values) in processor. The number of bits in a word is called the **word length** of the computer. Modern computers are 64-bit computers, meaning their word length is 64 bits. Earlier computers have 32-bit, 16-bit, and 8-bit word lengths.

The most fundamental digital symbols are bits, numbers, and characters. This section discusses three examples to show how to do binary-decimal number conversion, how to represent integers, and how to represent English characters. The focus is on **representation** of these symbols. Representation is the way the bits of a symbol are laid out when the symbol is stored in the computer memory. Once a symbol is properly represented, manipulation (operations on the symbol) often becomes obvious and intuitive.

2.1.1 *Conversions Between Binary and Decimal Number Representations*

The problem is to convert a number in binary representation to its decimal representation, and vice versa. It is helpful to have a table ready showing the corresponding values of binary and decimal bases, as shown in Table 2.1.

Example 2.1. $\mathbf{(110.101)_2 = (?)_{10}}$
$(110.101)_2 = 1\times2^2+1\times2^1+0\times2^0+1\times2^{-1}+0\times2^{-2}+1\times2^{-3} = 4+2+0.5+0.125 = (6.625)_{10}$.

Example 2.2. $\mathbf{(6.625)_{10} = (?)_2}$
We convert the integer part (6) and the fraction part (0.625) separately. The decimal value 6.625 is converted into the binary value 110.101.

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
8	4	2	1	0.5	0.25	0.125	0.0625	0.03125
	1	**1**	**0**	**1**	**0**	**1**		

Students show different tastes for this binary-decimal conversion problem and prefer different methods. There is no best conversion algorithm for all students. We will not formally describe a conversion algorithm. Instead, we use a more intuitive way of illustrating a conversion algorithm using the specific problem of converting 6.625 into 110.101.

Converting the integer part 6 into binary representation needs three steps. The conversion algorithm goes as follows. It uses a variable called the remainder.

- Initialize the remainder as 6. Look at Table 2.1.
- Start from the column with the largest decimal base that is less than or equal to 6. The matching column is column 4, not column 8 or column 2.
- Work from left to right, one column at a time.
  - Try to subtract the decimal base from the remainder, write down the result (1 if sufficient, 0 otherwise) and the remainder in parentheses.
  - When the remainder is 0, stop.

Table 2.1 Correspondence of binary and decimal bases

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
10000.	1000.	100.	10.	1.	0.1	0.01	0.001	0.0001	0.00001
16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

The binary number of the integer part 6 is 110. Details of the three steps follow.

Step 1: 6-4=2; sufficient, the new remainder is 2, write down 1(2).

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)							

Step 2: 2-2=0; sufficient, the new remainder is 0, write down 1(0).

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	**1 (0)**						

Step 3: As remainder is 0, stop. Note that the remaining bit of the integer part, i.e., column 1, is empty. This is understood to represent 0.

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	1 (0)	**0**					

Converting the fraction part 0.625 uses a similar algorithm. It needs four steps. Initially, let remainder be 0.625. Start from column 0.5 and work from left to right.

Step 4: 0.625-0.5=0.125; sufficient, the remainder is 0.125, write down 1 (.125).

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	1 (0)	0	**1 (.125)**				

Step 5: 0.125-0.25; insufficient, the remainder is 0.125, write down 0 (.125).

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	1 (0)	0	1 (.125)	**0 (.125)**			

Step 6: 0.125-0.125=0; sufficient, the remainder is 0, write down 1 (0).

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	1 (0)	0	1 (.125)	0 (.125)	**1 (0)**		

Step 7: As the remainder is 0, stop. The final result is $(6.625)_{10} = (110.101)_2$.

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1	**1**	**0**	**1**	**0**	**1**		

Table 2.2 Binary, decimal, and hexadecimal representations of natural numbers

Binary	Decimal	Hexadecimal
$2^3 2^2 2^1 2^0$	$10^1 10^0$	16^0
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Example 2.3. $(11.3)_{10} = (?)_2$
Use the same method to convert 11.3. This is an infinite process, corresponding to a binary number with an infinitely cyclic fraction. The final result is

$(11.3)_{10} = (1011.010011001\ldots\ldots)_2.$

Note that the largest decimal base less than 11 is 8. The conversion result after the 13th step is shown below.

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
8	4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (3)	0 (3)	1 (1)	1 (0)	0 (.3)	1(.05)	0 (.05)	0 (.05)	1 (.01875)

2^{-6}	2^{-7}	2^{-8}	2^{-9}
0.015625	0.0078125	0.00390625	0.001953125
1 (.003125)	0 (.003125)	0 (.003125)	1 (.001171875)

Equipped with the above conversion method, we can represent all natural numbers, i.e., 0 and positive integers, in the binary notation. In addition, we use a base-16 notation called **hexadecimal** representation, as shown in Table 2.2.

The hexadecimal representation is a base-16 notation, meaning a digit has 16 values, from 0, 1, ... to 15. To avoid confusion, we replace the six 2-digit symbols 10, 11, 12, 13, 14, 15 with six 1-digit symbols A, B, C, D, E, F. Note that hexadecimal digit symbols can also be written in small case: a, b, c, d, e, f. They represent the same values as A, B, C, D, E, F.

Each hexadecimal digit represents four bits. Converting a binary number to a hexadecimal number is easy: we simply partition the binary number into 4-bit groups, starting from the least significant bit, and then convert each 4-bit group into a hexadecimal digit according to Table 2.2.

For instance, to represent the decimal value 63 in an 8-bit binary representation, we have 63 = 00111111. Partitioning it into 4-bit groups, we have 0011 1111 = $3F_{16}$. The hexadecimal representation $3F_{16}$ is sometimes simply written as 3F when there is no confusion. In computer programs, we often write 0x3F, where **0x** denotes hexadecimal representation. Some computers differentiate capital or small cases, such that $3F_{16}$ is written as 0x3f or 0X3F.

Having fewer numbers of digits, the hexadecimal representation is often easier for humans to understand and use than binary representation.

2.1.2 *Representing Integers in Two's Complement Representation*

The above examples seem to suggest a natural way to represent natural numbers and integers. If we have n bits, we can precisely represent all 2^n natural numbers in the interval [0, 2^n-1], such that binary 0…00 represents decimal 0, binary 0…01 represents 1, and binary 1…1 represents 2^n-1. When n=8, we can represent all the 256 natural numbers in the interval [0, 255], where 00000000 = 0, 00000001 = 1, …, and 11111111 = 255. This is called the **unsigned integer** representation.

How about integers? A straightforward method, called the **simple signed integer** representation, is to use the leftmost bit for the sign bit, and the remaining n-1 bits for the absolute value. Thus, 8 bits are enough to represent integers in the interval [-127, 127], as 2^7=128. However, this intuitive representation has problems, as the following example shows.

63 = 00111111, 64 = 01000000, (-63) = 10111111, (-64) = 11000000.
63 + 64 = 00111111 + 01000000 = 01111111 = 127 (correct)
(-63) + (-64) = 10111111 + 11000000 = 11111111 = (-127) (correct)
63 + (-63) = 00111111 + 10111111 = 11111110 = (-126) (wrong!)

A smarter representation is called **two's complement** representation. Zero and positive numbers are represented in the usual way. A negative number is represented by its two's complement: (1) finding the binary representation of its absolute number, (2) bit-wise inverting the binary representation, and (3) adding 1 to the inverted number. The negative integer (-63) is represented as 11000001, because

1. the binary representation of the absolute value of (-63) is 63=00111111,
2. bit-wise inverting 00111111 yields 11000000, and
3. adding 1 yields 11000000+00000001 = 11000001.

This smarter representation solves the above problem. Let us verify it by redoing the arithmetic, noting two details when doing addition: (1) the sign bits are treated the same as the other bits, and (2) the carry over the sign bit is ignored.

63 = 00111111; 64 = 01000000; (-63) = 11000001, (-64) = 11000000
63 + 64 = 00111111 + 01000000 = 01111111 = 127 (correct)
(-63) + (-64) = 11000001 + 11000000 = 10000001 = (-127) (correct)
63 + (-63) = 00111111 + 11000001 = 00000000 = 0 (correct!)

A bit-by-bit process is shown below. Note that the carry bit over the sign bit is ignored (boldfaced).

```
63 + (-63) =        00111111 + 11000001 = 100000000 = 00000000₂ = 0₁₀
                    11000001
The carry bits     11111111
The result bits    100000000 = 00000000₂ = 0₁₀
```

2.1.3 *Representing English Characters: The ASCII Characters*

Any finite set of symbols can be represented by one or more bits. Any symbols, not just numbers.

Suppose a symbol set has more than 2^{n-1} but no more than 2^n symbols. A straightforward method of representation is to use n-bit numbers, 2^n of them in total, to represent the symbol set, such that each n-bit number represents a distinct symbol of the set.

A basic format for representing English characters is **ASCII** (American Standard Code for Information Interchange), which uses one **byte** (8 bits), as shown in Fig. 2.2. Actually, only 7 bits ($D_6D_5D_4$ $D_3D_2D_1D_0$) are used to represent characters, the highest bit (D_7) is used for other purpose, such as extension or error detection. So D_7 is always 0 in Fig. 2.2.

Seven bits have 128 combinations and can represent 128 symbols. Of these 128 combinations, 33 combinations (the first 32 and the last combinations) are used to represent control characters, such as carriage return, escape, and delete. The remaining 95 combinations are used to represent “normal” characters in a usual English text, such as characters in the alphabet (A, . . ., Z, a, . . ., z), decimal numbers (0, . . ., 9), various punctuation and other symbols (+, !, @, #, $, %, etc.).

The value of a character in Fig. 2.2 is also called the **ASCII encoding** of that character, also known as *ASCII code* or *ASCII value*. The value is an 8-bit unsigned integer value, and could be displayed in binary, decimal, or hexadecimal formats. Since the leftmost bit is always zero, the value of an ASCII character is between 0 (for the null character NUL) and 127 (for the delete character DEL).

$D_7D_6D_5D_4$ / $D_3D_2D_1D_0$	0000	0001	0010	0011	0100	0101	0110	0111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Fig. 2.2 Representation of ASCII Characters

For instance, from Fig. 2.2, we can see that the ASCII encoding for letter X is $01011000_2 = 88_{10}$. The ASCII encoding for the plus sign ‘+’ is $00101011_2 = 43_{10}$. The ASCII encoding for escape character ESC is $00011011_2 = 27_{10}$.

The character string “Alan Turing” contains 11 characters, one of which is a space (SP). This character string’s ASCII encoding is “Alan Turing” = [65, 108, 97, 110, 32, 84, 117, 114, 105, 110, 103].

Three ASCII characters need special mention, i.e., null (NUL), space (SP), digit 0. Some students find them confusing, probably because they all intuitively indicate some forms of emptiness. But they are quite different characters. In particular, note that the ASCII encoding for digit 0 is not 0, but 48. The ASCII value 0 is used for the null character NUL. We contrast the ASCII encodings for these three characters below and mark them in Fig. 2.2.

ASCII value for the null character NUL:
$00000000_2 = 0_{10}$
ASCII value for the space character SP:
$00100000_2 = 32_{10}$
ASCII value for the digit 0 character:
$00110000_2 = 48_{10}$

2.2 Programs as Symbols

For students new to programming, it helps to write and run a number of simple programs with increasingly complex structures. Deliberated errors are included in some programs to show and debug compiling errors and runtime errors.

2.2.1 A Number of Simple Programs

It is common practice to ask students to write their first program to output some form of "Hello, world!". Figure 2.3 starts with an even simpler program and then adds several more programs, some of them containing errors. The point is to familiarize the students with the edit-compile-execute process.

The null.go program is correct but does nothing. The program hello.go is correct and outputs hello!. The program hello-1.go contains compiling errors. The screen output of each program's compile-execute process is shown below. The ">" symbol is the **command-line prompt**.

```
> go build null.go   ; Compile null.go into an executable file null
> ./null         ; Execute null
>        ; The program does nothing and returns to shell
```

```
package main                // declare main package of the program
func main() {               // declare main function of the program
}                           // the body of the function is empty
```

(a)

```
package main
import "fmt"                // import a library package "fmt"
func main() {
   fmt.Println("hello!")    // which is used here to print out things
}
```

(b)

```
package main
func main { } (                         // wrong parentheses are used
)
```

(c)

Fig. 2.3 Some simple programs. (**a**) The simplest Go program null.go which is correct but does nothing. (**b**) A correct program hello.go which outputs hello! (**c**) A wrong program hello-1.go which produces compiling error

```
> go build hello.go   ; Compile hello.go into an executable file hello
> ./hello        ; Execute hello
hello!       ; The program outputs hello!
>      ; The program finishes and returns to shell
```

The two steps of the compile-execute process can be combined into one step.

```
> go run hello.go     ; use "run" instead of "build"
hello!
> go run hello-1.go
# command-line-arguments
.\hello-1.go:2:6: missing function body
.\hello-1.go:2:11: syntax error: unexpected {, expecting (
>
```

Actually, three parties are involved in executing the above commands and programs: the human user, the command-line environment of the operating system called the **shell** environment, and the rest of the computer. The shell provides a user interface for the user to enter a command and see the execution result of the command. The shell also interprets (executes) a command and generates the result of execution. Recall that commands are also programs.

During these processes, a program needs to do three things besides executing internal instructions: accepts input, produces output, and produces error output. Now we encounter a problem: where is the source/destination? Accept input from where? Where is the produced output sent? Produce error output to which device? Modern computers have a default answer to these questions, unless specified by the user otherwise:

- Accept input from the **Standard Input** device, usually the keyboard device. It is often denoted by a name such as StdIn, stdIn, or stdin, in programs.
- Send output to the **Standard Output** device, usually the display screen. It is often denoted by a name such as StdOut, stdOut, or stdout in programs.
- Send error output to the **Standard Error** device, usually the display screen. It is often denoted by a name such as StdErr, stdErr, or stderr in programs.

Sometimes, the source/destination object to input or output is a file stored in the hard disk, but we use a name to refer to the file. We go through the above simple programs again, paying attention to how the standard input, output, and error output behave.

```
>go build null.go   ; Shell gets input from StdIn, with file name null.go
>       ; No output sent to StdOut. A file null sent to disk.

> ./null     ; Shell gets input from StdIn, with file name null
>       ; No output sent to StdOut.
```

```
 > ./hello     ; Shell gets input from StdIn, with file name hello
 hello!       ; Program hello sends output "hello!" to StdOut
 >      ; The program finishes and returns to shell

 > go build hello-1.go   ; Shell gets input from StdIn, with file name
hello-1.go
 # command-line-arguments       ; Error messages to StdErr
 .\hello-1.go:2:6: missing function body
 .\hello-1.go:2:11: syntax error: unexpected {, expecting (
 >
```

We can use the symbol '<' to redirect standard input, and the symbol '>' to redirect standard output, respectively. For instance, the following command

```
> ./hello > helloResult
>
```

sends nothing to StdOut, because the result is redirected to file helloResult.

2.2.2 Programs Relating Character Strings to Integers

We can better understand the most basic digital symbols, i.e., numbers and characters, by writing three programs: symbols.go, name_to_number-0.go, and name_to_number.go. The basic digital symbols manifest as simple **data types** and their representations, such as integer, array, and character string types, and decimal, hexadecimal, binary, and character representations. These representations are specified using different **formatting verbs** in a fmt.Printf statement.

The first program symbols.go shows these different representations of the same value 63. The program generates four different screen outputs 63, 0x3F, 111111, and '?', by using four different formatting verbs %d, %X, %b, and %c, respectively.

Of the four formatting verbs, the character verb %c is the most basic. The reason is that the display screen only prints out one character at a time. Printing out decimal value 63 by the %d verb is actually done by using %c twice to output ASCII characters '6' and '3'. The last three fmt.Printf statements each try to implement the %d verb functionality using only %c by outputting a string of two characters 6 and 3 (Fig. 2.4).

The fmt.Printf("String: %c%c\n",63) statement naively uses two %c verbs for the two characters 6 and 3. It forgets that 63 is one value. The next statement separates 63 into two values 6 and 3 before printing. It fails because 6 and 3 are the ASCII code value for control characters ACK and EXT, displayed as ═ and ╙, respectively. The last statement remedies this by adding '0', which represents character digit 0 and has a value of 48. Thus, 6+'0'=54 and 3+'0'=51, respectively, which are the correct ASCII code values corresponding to characters 6 and 3.x

```
package main
import "fmt"
func main() {
  fmt.Printf("Decimal: %d\n",63)
  fmt.Printf("Hex: %X\n",63)
  fmt.Printf("Binary: %b\n",63)
  fmt.Printf("Character: %c\n",63)
  fmt.Printf("String: %c%c\n",63)
  fmt.Printf("String: %c%c\n",6,3)
  fmt.Printf("String: %c%c\n",6+'0',3+'0')
}
```

(a)

```
> go run symbols.go
Decimal: 63              ; decimal representation of value 63
Hex: 3F                  ; hexadecimal representation of value 63
Binary: 111111           ; binary representation of value 63
Character: ?             ; ASCII character corresponding to 63
String: ?%!c(MISSING)    ; error, 63 is one value, not for two characters
String: ═ ╙              ; error, output control characters ACK and EXT
String: 63               ; correctly output two characters 6 and 3
>
```

(b)

Fig. 2.4 Program symbols.go and its output. (**a**) Program symbols.go. (**b**) Output by executing program symbols.go

The second program computes the *student code* from a student name, represented as a string of ASCII characters. More specifically, program name_to_number-0.go in Fig. 2.5 outputs the sum of the ASCII code values of the eleven characters in the student name string "Alan Turing". Students are suggested to read the material through to Fig. 2.7, which will make the material easier to understand.

This example introduces four new types of digital symbols: **variable**, **array**, **string**, and **loop**. Variables represent those digital symbols the values of which may change during a program's execution. In contrast, a **constant** symbol does not change its value. Variables should be declared before using. The statement

```
var name string = "Alan Turing"
```

declares a variable: its name is name, its data type is string (a byte array), and its initial value is "Alan Turing". Any digital symbol has these three aspects: name, type, and value. The above declaration statement can be shortened to

```
name := "Alan Turing",
```

which is valid within a code block between { and }.

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0                          // sum is type int, i.e., 64-bit integer
   for i := 0; i < 11; i++ {         // i  is type int
     sum = sum + int(name[i])
   }
   fmt.Printf("%d\n", sum)
}
```

(a)

```
> go run name_to_number-0.go
1045
>
```

(b)

Fig. 2.5 Program name_to_number-0.go and its output. (**a**) Source code of program name_to_number-0.go. (**b**) Screen output by executing program name_to_number-0.go

An **array** is a variable with 0 or more elements of the same data type, as illustrated in Fig. 2.6. A string variable is an array such that its elements are of type **byte** and their values can only be initialized but not altered. The data type byte is also called **uint8**, i.e., 8-bit unsigned integer that can have a value from 0 to 255.

The character **string** "Alan Turing" contains 11 elements, represented in a computer memory as an array: "Alan Turing" = [65, 108, 97, 110, 32, 84, 117, 114, 105, 110, 103]. As the initial value, this string is assigned to an array variable called name. The array's **length** is 11, the number of the array elements. The length of array name can be found by calling a system-provided function len(name).

We use name[i] to specify the i-th element of array name, where i is called the array **index**. The index's value starts from 0 and increments up to len(name)-1, or 11-1=10. Thus,

```
   name[0]='A'=65,     name[1]='l'=108,   name[2]='a'=97,
   name[3]='n'=110,     name[4]=' '=32,     name[5]='T'=84,
   name[6]='u'=117,   name[7]='r'=114,    name[8]='i'=105,   name[9]
='n'=110,   name[10]='g'=103.
```

Note that each array element is a variable of type byte (8-bit unsigned integer). It can hold the ASCII encoding of a character. In the above string example, name [0] holds English letter A, which has ASCII encoding 65. We need to pay attention to name[4], which holds the space character ' ' (SP), with ASCII encoding 32.

Program name_to_number-0.go produces the sum of these eleven numbers, to output 1045. That is: 65+108+97+110+32+84+117+114+105+110+103 = 1045. The program does this summation by the following **for loop** statement:

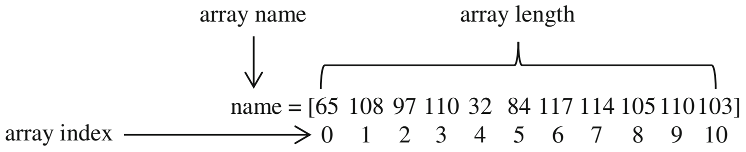


Fig. 2.6 Illustration of an array variable, called name, after the declaration statement var name string = "Alan Turing"

```
for i := 0; i < 11; i++ {          // 0 ≤ i < 11; increment i
    sum = sum + int(name[i])          by 1 at each iteration
 }
```

Start with i = 0. Repetitively execute the loop body until i ≥ 11. At each repetition (called *iteration*), increment i by 1. This is what i++ means.

The **loop body** is the code block between { and } of the for loop. Here, the loop body is the **assignment** statement sum = sum + int(name[i]), which assigns the value of the right-side **expression** sum + int(name[i]) to the left-side variable sum.

In other words, the for loop statement is a shorthand notation for executing the loop body 11 times, equivalent to the following 11 lines of code:

```
sum = sum + int(name[0])
sum = sum + int(name[1])
sum = sum + int(name[2])
sum = sum + int(name[3])
sum = sum + int(name[4])
sum = sum + int(name[5])
sum = sum + int(name[6])
sum = sum + int(name[7])
sum = sum + int(name[8])
sum = sum + int(name[9])
sum = sum + int(name[10])
```

This for loop accumulatively adds up the 11 elements of array name, and puts the result in the integer variable sum. Note that before the for loop, sum is already initialized to 0 by the sum = 0 statement, as shown in Fig. 2.5a.

Some students may find the expression sum + int(name[i]) strange. Why not simply write the expression as sum + name[i]?

The lecturer can deliberately make a mistake here by showing what error will occur if we use expression sum + name[i]. Two key ideas can be revealed: (1) only values of the same data type can be added (operated); and (2) if an operation involves values of different types, a **type cast** operation can be used to convert a value into the desired type.

The four values involved in the "sum = sum + int(name[0])" assignment statement are shown in Table 2.3. Before executing the statement, variable sum (right-side) holds a 64-bit integer value 0, and name[0] holds an 8-bit unsigned integer value 65. After execution, sum (left-side) holds a 64-bit integer of value 65.

Table 2.3 Type casting makes an operation on values of different types possible

Value name	Binary representation
sum (right-side)	0000000000000000000000000000000000000000000000000000000000000000
name[0]	01000001
int(name[0])	0000000000000000000000000000000000000000000000000000000001000001
sum (left-side)	0000000000000000000000000000000000000000000000000000000001000001

In the right-side expression sum + int(name[0]), variable sum is of type int (64-bit integer), and name[0] is of type byte (8-bit unsigned integer). They cannot be added. We need the type cast operation int(. . .), to convert name[0], a value of type byte, to a value int(name[0]) of integer type, and then add to integer variable sum. The type cast operation int(name[0]) pads the 8-bit value 01000001 of name[0] into a 64-bit value, adding 56 0's to the left.

The last statement of program name_to_number-0.go is an output statement. It prints out the value of sum by using an fmt.Printf statement with the formatting verb %d. That is, output the value of sum in decimal representation.

Example 2.4. Realizing a High-Level Formatting Verb with a Basic Verb
What if we only have the %c formatting verb? A challenge to students is to implement formatting verb %d in fmt.Printf("%d\n", sum) by using only the basic formatting verb %c. This is done by the third program name_to_number.go, which demonstrates how to realize a more complex operation (the %d verb) via elementary operations (the %c verb), as shown in Fig. 2.7. The functionality of the single-line

```
package main
import "fmt"
func main() {
    var name string = "Alan Turing"
    sum := 0
    for i := 0; i < 11; i++ {
       sum = sum + int(name[i])
    }
    var sum_bytes [4]byte
    var j int
    for j = 3; sum != 0; j-- {
       sum_bytes[j] = byte(sum%10) + '0'
       sum = sum / 10
    }
    fmt.Printf("%c", sum_bytes[0])
    fmt.Printf("%c", sum_bytes[1])
    fmt.Printf("%c", sum_bytes[2])
    fmt.Printf("%c", sum_bytes[3])
    fmt.Printf("\n")
}
```

Fig. 2.7 Program name_to_number.go and its output

statement fmt.Printf("%d\n", sum) in name_to_number-0.go is realized by the eleven lines of code (marked in red) in name_to_number.go.

Writing name_to_number.go as a personalized program different for each student is left as a programming exercise. Let us see how "1045" in Fig. 2.4b is printed out by noticing the following. In Go notation, sum%10 is a modulus operation, i.e., sum mod 10. It generates the remainder when sum divides 10. For instance, 86%10 generates 6. Expression sum / 10 is an integer division, and the result is rounded to integer. For instance, 86/10 = 8, not 8.6.

The for j loop is equivalent to the following sequence of statements:

```
 sum_bytes[3]=byte(sum%10)+'0' // sum_bytes[3]=byte(1045%10)+'0'
(='5')
 sum = sum / 10       // sum=1045/10  (=104)
 sum_bytes[2]=byte(sum%10)+'0' // sum_bytes[2]=byte(104%10)+'0'
(='4')
 sum = sum / 10       // sum=104/10   (=10)
 sum_bytes[1]=byte(sum%10)+'0' // sum_bytes[1]=byte(10%10)+'0'
(='0')
 sum = sum / 10       // sum=10/10    (=1)
 sum_bytes[0]=byte(sum%10)+'0' // sum_bytes[0]=byte(1%10)+'0'
(='1')
 sum = sum / 10       // sum=1/10     (=0)
```

The final print statement:

```
fmt.Printf("\n")
```

changes to the next line (new line), to make a clean printout.

2.2.3 *Good Programming Practices*

This UKA unit introduces students to good programming practices. The resulting code might be longer, but is easier for humans to understand, use, and maintain. We illustrate five such practices by revising the code in Fig. 2.7. Modifying or updating programs to improve software quality is called software maintenance, meaning to **maintain the code**. The updated code, shown in Fig. 2.8, has several differences from and improvements over the original code in Fig. 2.7.

- *Use descriptive names* for variables and constants. The new code uses more descriptive studentName and sumBytes, both in camel notation, to replace the less descriptive names: name and sum_bytes.
- *Avoid magic numbers*. The old code contains three magic number, 11, 4, 3, in order of appearance. **Magic numbers** are numbers directly appearing in code without context or explanation. A fellow programmer cannot understand what the

```
package main
import "fmt"
const studentName          = "Alan Turing"
const maxCodeLength        = 4        // student code has at most 4 digits
func main() {
   sum := 0
   for i := 0; i < len(studentName); i++ {      // add up studentName to sum
     sum = sum + int(studentName[i])
   }
   var sumBytes [maxCodeLength]byte       // array to hold characters of sum
   var j int
   for j = len(sumBytes) - 1; sum != 0; j-- { // extract each digit from sum
     sumBytes[j] = byte(sum%10) + '0'
     sum = sum / 10
   }
   var k int
   for k = j + 1; k < len(sumBytes); k++ {    // print each digit of sum
     fmt.Printf("%c", sumBytes[k])
   }
   fmt.Printf("\n")
}
```

Fig. 2.8 Program name_to_number-1.go with coding practice improvements

numbers indicate and why they have such values. The new code replaces 11, 4, 3 by three descriptive expressions len(studentName), maxCodeLength, and len (sumBytes) – 1, respectively. The updated code has no more magic number.

- *Avoid repetitive code*. The updated code uses an abstraction, the for k loop, to replace the repetitive code of four print statements.
- *Put constant definitions up front*. The updated code differentiates constants from variables. It puts the two constant definitions up front, i.e., in one place at the beginning of the code. If we want to print out the code value for another student, e.g., "Gordon Moore" instead of "Alan Turing", we only need to go to this single conspicuous place to modify the code.
- *Use comments to document the code*. Five lines of comments are added to help users understand the code. Such comments are called **documentation** of a program. Documentation is not necessary for a program to execute. However, proper documentation improves the understandability of code.

2.2.4 Using Dynamic Programing to Compute Fibonacci Number F(50)

This UKA unit serves two purposes: to show that solving a larger-scale problem may need a smarter algorithm; and to show that smarter algorithms may need new program structures (such digital symbols are often called programming language **constructs**). It is done by writing two programs to compute larger Fibonacci numbers in two methods, recursive and dynamic programming. Two new constructs are introduced: **function** and **slice**. Figure 2.9 contrasts these two programs fib-50.go and fib.dp-50.go.

```
package main
import "fmt"
func main() {
  fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) int {
  if n == 0 || n == 1 {
    return n
  }
  return fibonacci(n-1)+fibonacci(n-2)
}
```

(a)

```
package main
import "fmt"
func main() {
  fmt.Println("F(50)=", fibonacci(50))
}
func fibonacci(n int) int {
  if n == 0 || n == 1 {
    return n
  }
  var fib []int = make([]int, n+1)    // make a slice fib
  fib[0] = 0                          // initialize fib[0] and fib[1]
  fib[1] = 1
  for i := 2; i <= n; i++ {           // iteratively compute fib[i]
    fib[i] = fib[i-1] + fib[i-2]
  }
  return fib[n]
}
```

(b)

Fig. 2.9 The recursive and dynamic programming programs to compute Fibonacci numbers. (**a**) Recursive fib-50.go. (**b**) Dynamic programming fib.dp-50.go

A **function** is a sub-program to be used (*called*) by other statements in a program. An example function definition starts with the keyword func:

```
func fibonacci(n int) int { ... }
```

It consists of four parts: (1) a *function name* fibonacci, (2) an input *parameter* n of integer type, (3) a *return value* of type int, and (4) a *function body* which is a sequence of statements enclosed between the curly brackets { and }.

This fibonacci function is used (called) in the statement

```
fmt.Println("F(50)=", fibonacci(50))
```

by a function call fibonacci(50), where the parameter n assumes a value of 50. A function can call itself. This recursive call is present in fib-50.go.

Program fib-50.go is almost the same as fib-10.go in Example 1.1. The only difference is that we are computing a larger Fibonacci number F(50), instead of F(10). The lecturer can compare these two programs by noticing their execution time. The fib-50.go program, although very intuitive to the mathematic definition, is painfully slow. It takes 3 minutes to output the result F(50) = 12586269025. The fib.dp-50.go program is much faster, taking just a second. The reason is that the second program utilizes a smarter algorithmic method called **dynamic programming**: intermediate results F(i-1) and F(i-2) are memorized and accessed to compute F(i), as demonstrated in the loop structure containing statement fib[i] = fib[i-1] + fib[i-2]. This method avoids repetitions in computing F(i) multiple times in fib-50.go.

To support this memorization, a new data type called **slice** is used in the fib.dp-50.go program. The statement

```
var fib []int = make([]int, n+1)
```

declares a slice variable fib which points to an underlying array of n+1 elements of type int. The length of the slice is the length of the underlying array, which can be found by calling len(fib). The ith element of the slice is accessed via fib[i], where the index i starts from 0 to n, namely len(fib)-1. The make function is a system provided function, which creates and returns a slice with an underlying array of n+1 elements of type int. All n+1 elements of the slice are initialized with the zero value (Fig. 2.10).

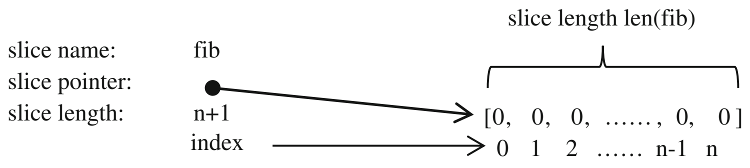


Fig. 2.10 Illustration of a slice variable fib, after statement var fib []int = make([]int, n+1)

After making the slice fib, the program first initializes the first two elements fib [0] and fib[1], and then iteratively computes fib[i], such that all elements from fib [0] to fib[50] are computed exactly once. The sequence of execution steps is like the following:

```
fib[0] = 0
fib[1] = 1
fib[2] = fib[1] + fib[0]      // fib[2] = 1 + 0 = 1
fib[3] = fib[2] + fib[1]      // fib[3] = 1 + 1 = 2
...
...
fib[48] = fib[47] + fib[46]    // fib[48] = 2971215073 + 1836311903
                                          = 4807526976
fib[49] = fib[48] + fib[47]    // fib[49] = 4807526976 + 2971215073
                                          = 7778742049
fib[50] = fib[49] + fib[48]    // fib[50] = 7778742049 + 4807526976
                                          = 12586269025
return fib[50]        // return 12586269025
```

Note that every newly computed Fibonacci value is stored (memorized) in slice element fib[i] and later referenced. No Fibonacci value is computed more than once.

2.3 Computer as a Symbol-Manipulation System

The example of computing Fibonacci numbers shows that symbol manipulation processes embodied in programs need the support of computer systems, to realize basic arithmetic-logic operations, variable, function, loop, array, and slice.

We introduce a general model of computers in this section. It is called the **stored program architecture** or stored program model, also known as the von Neumann model or **von Neumann architecture**. We will use these terms interchangeably, with this historical footnote.[1]

A computing system usually has three layers: hardware, system software, and application software, as illustrated in Fig. 2.11. Students so far have used the High-Level Language interface. This section introduces a low-level interface, i.e., computer **instructions**, to see how computers work. Most computer hardware today adopts a stored-program architecture with the following five characteristics.

[1] Although the term *von Neumann architecture* is widely used, it is controversial. A reason is that this term comes from a manuscript written by John von Neumann in 1945 with the title *First Draft of a Report on the EDVAC*. The original manuscript did not list any author. Herman Goldstine, a US Army officer overseeing the ENIAC project, circulated the report with only von Neumann's name on it. Some computer pioneers argued that key ideas in the report, including the stored program concept, were not proposed by von Neumann. Some books in computer architecture use terms such as "stored-program architecture", instead of the term "von Neumann architecture". See Bibliographic Notes for details.

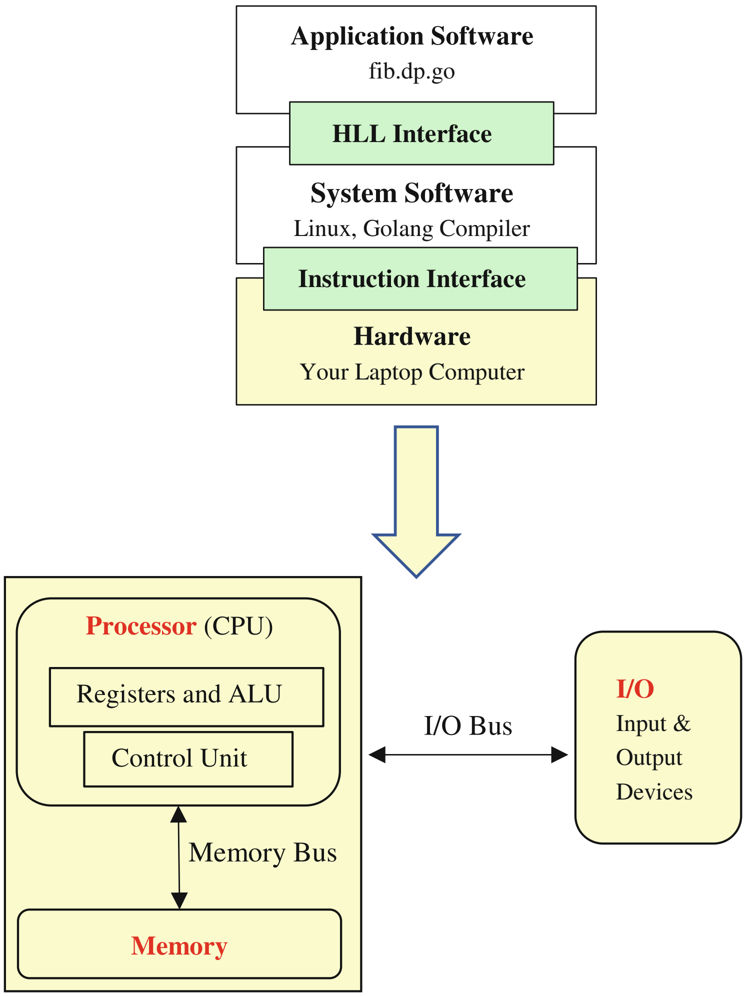


Fig. 2.11 The stored-program model of computers, also known as the von Neumann model

- **Binary**. Data and instructions use binary representations.
- **P-M-I/O**. The computer hardware is comprised of three interconnected components: **processor**, **memory**, and **I/O devices**.
  - The processor is also called **CPU**, for central processing unit. It executes instructions using an arithmetic logic unit (**ALU**) and a small number of general-purpose registers, under the control of a control unit. A modern processor may also contain other processing units, such as graphics processing unit (GPU) and machine learning processing unit.

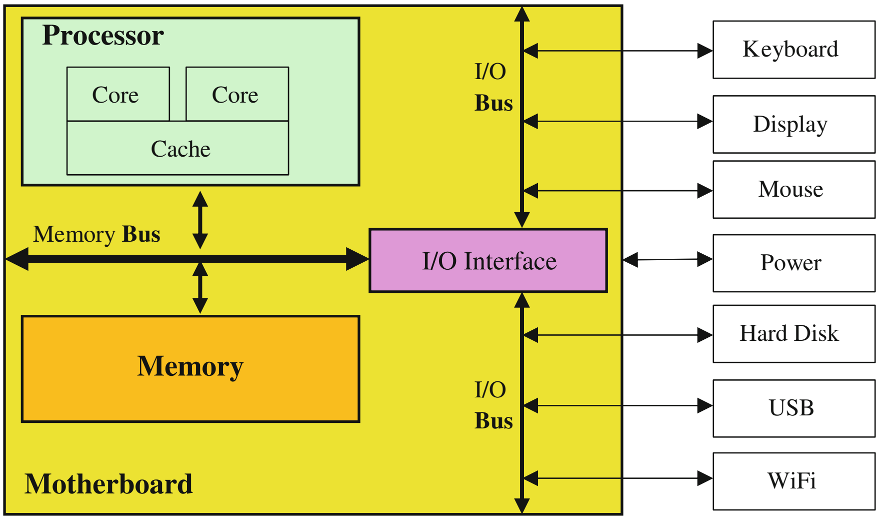


Fig. 2.12 Illustration of how main components are organized to form a computer

- The memory is also called **main memory**, accessed by CPU with an instruction. Registers may be considered special memory cells in CPU.
- I/O devices include hard disk, keyboard, mouse, display, printer, etc.

- **Stored program**. Both programs and data are stored in the memory and accessed by processor.
- **Instruction driven**. The computer changes its state (the contents of memory and registers) only when an instruction is executed.
- **Serial execution**. A computational process is a serial-execution process. Any program is executed by automatically executing one instruction after another.

2.3.1 A Glimpse Inside a Computer

We can use the von Neumann model to look inside a computer, to see how the components are organized and interconnected to form the hardware of a computer. This more detailed inside organization is shown in Fig. 2.12.

The components in the right part of Fig. 2.12 are I/O devices. Processor and memory are in the left part. They are interconnected by the memory bus and the I/O bus. A popular I/O bus is the PCIE bus, for the Peripheral Component Interconnect Express bus. An I/O Interface circuitry bridges the memory bus and the I/O bus. The power unit (such as a battery) is also shown.

Motherboard is the main printed circuit board which provides a physical substrate to host the memory bus, the I/O bus and the I/O Interface. All processor,

Table 2.4 Parameters of a desktop computer according to von Neumann model

Processor	Intel Core i5-4460 CPU @3.20 GHz, 6 MB cache	
Memory	16 GB main memory	
I/O devices	Storage	640 GB hard disk
	Keyboard	Standard Dell keyboard
	Display	2560×1440 resolution
	Mouse	Optical mouse
	Network	100 Gbps Ethernet, 100 Mbps WiFi

memory, I/O Interface microchips, and other circuitry interfacing the I/O devices, are soldered on or plugged into the motherboard. The processor is a modern **multicore** processor, capable of parallel processing. Each of the two cores is a CPU. A small but fast memory, called **cache**, is also present in the processor.

It helps for each student to inventory his/her personal computer, e.g., laptop computer, to make the above concepts more concrete and vivid. A hands-on exercise is to list the main components of the computer according to the von Neumann model, in the form of a table similar to Table 2.4, which contains data from a desktop computer. This is an incomplete list, but already can lead to some interesting questions. For instance, students have asked: why is a hard disk an I/O device? The hard disk and the memory both stores data. Why do we distinguish them?

2.3.2 A Step-By-Step Process on a von Neumann Computer

To see why and how a computer is a symbol manipulation system, in this UKA unit the students are asked to meticulously go through 16 steps of an example code, where each step executes an instruction and forms a computer **state transition**.

The state of a computer at any time is comprised of the content of the main memory and the content of the registers in the processor. We ignore the I/O devices in this example. We consider only three types of registers here:

- *General-purpose registers*, denoted as R0, R1, R2.
- *Special-purpose registers*, two of which are used here.
  - Status register FLAGS, which holds a set of status flag bits, to denote the status of an instruction's execution. Examples include whether the result is zero, positive or negative, whether there is an overflow, etc.
  - Program counter (PC), which holds the address of the instruction to be executed next.

We show a step-by-step example how this **Fibonacci Computer** executes the dominant part of the fib.dp-50.go program, i.e., the for loop structure:

```
for i := 2; i < 51; i++ {     // n+1 is 51 for F(50)
  fib[i] = fib[i-1] + fib[i-2]
}
```

Code snippets of the Go language program (HLL interface) and the corresponding **assembly language** code (instruction interface) are shown below side by side, to highlight their correspondences. Assembly language code is a sequence of instructions in human understandable form, instead of a string of 0's and 1's. The two forms of code should be studied referring to the 17 tables in the following pages, which show the state transitions of the computer hardware.

```
fib[0] = 0                              MOV 0, R1
                                        MOV R1, M[R0] //R0=12 initially
fib[1] = 1                              MOV 1, R1
                                        MOV R1, M[R0+8]
for i = 2; i < 51; i++ {                MOV 2, R2      // i=2
fib[i] = fib[i-1] + fib[i-2]      Loop: MOV 0, R1      // label Loop
                                        ADD M[R0+R2*8-16], R1
                                        ADD M[R0+R2*8-8], R1
                                        MOV R1, M[R0+R2*8-0]
                                        INC R2         // i++
                                        CMP 51, R2     // i < 51?
}                                       JL Loop        // if Yes, goto Loop
```

Table 2.5 shows the initial state of the computer hardware. The binary code of the twelve instructions is stored in the main memory from address 0 to address 11, each instruction occupying one byte. In this example we assume the computer has only these instructions. In general, the set of instructions a computer has available is called the **instruction set** of that computer, which forms the instruction interface.

Note that address 5 has a label Loop, indicating the starting address of the loop in the code. Addresses 12~419 hold data array fib, e.g., addresses 12~19 for fib[0], 20~27 for fib[1], etc. Each array element fib[i] is a 64-bit integer, needing 8 bytes.

Five registers are shown. FLAGS is the **program status register**, which holds the status after an instruction's execution, such as whether the compare instruction (CMP) returns <, =, or >. PC is **program counter**, which holds the address of the next instruction to be executed. PC = 0 when the program initially starts. FLAGS and PC belong to the Control Unit in Fig. 2.12.

The example also shows three general-purpose registers R0, R1, and R2. Register R0 is used as a **base register**, which holds the base address of array fib, with an initial value of 12, i.e., the address of fib[0]. Register R1 is used as an **accumulator**, to hold the value of repetitive additions. Register R2 is used as an **index register**, to hold the value of array index i in fib[i]. The address of an array element fib[i] is calculated by **address = base + index*8 + offset**. The number 8 here is due to 8 bytes in a 64-bit integer.

Thus, to realize fib[i] = fib[i-1] + fib[i-2], we need the following instructions:

Table 2.5 Initial state: PC=0, R0=12; instructions addresses range 0~11, data addresses range 12~419. M[k] denotes the memory cell at address k

CPU contents		Memory contents		
Register	Value	Address	Instruction	Comments
FLAGS		0	MOV 0, R1	0→R1
PC	**0**	1	MOV R1, M[R0]	R1→M[R0]
R0	**12**	2	MOV 1, R1	1→R1
R1		3	MOV R1, M[R0+8]	R1→M[R0+8]
R2		4	MOV 2, R2	2→R2
R0: base register		5 Loop	MOV 0, R1	0→R1
Initial value=12		6	ADD M[R0+R2*8-16], R1	R1+ M[R0+R2*8-16] → R1
		7	ADD M[R0+R2*8-8], R1	R1+ M[R0+R2*8-8] → R1
R1: accumulator		8	MOV R1, M[R0+R2*8-0]	R1→ M[R0+R2*8-0]
R2: index register		9	INC R2	R2+1→R2
Address=base +index*8+offset		10	CMP 51, R2	Compare R2 to 51, status→FLAGS
		11	JL Loop	If FLAGS is "<", Loop→PC
fib[i-2]'s address		12		fib[0]
=R0+R2*8 -16		20		fib[1]
		28		fib[2]
fib[0]'s address		36		fib[3]
=12+2*8-16=12		……		……
		412		fib[50]

```
MOV 0, R1           // initialize accumulator R1 to 0
ADD M[R0+R2*8-16], R1   // R1+ fib[i-2] → R1
ADD M[R0+R2*8-8], R1     // R1+ fib[i-1] → R1
MOV R1, M[R0+R2*8-0]     // R1 → fib[i]
```

When i=3, we compute fib[3] = fib[2] + fib[1]. We have base=12, index=3, and

```
fib[3] = fib[i-0]; its address is R0+R2*8-0=12+3*8-0=36; offset is 0
fib[2] = fib[i-1]: its address is R0+R2*8-8=12+3*8-8=28; offset is -8
fib[1] = fib[i-2]: its address is R0+R2*8-16=12+3*8-16=20; offset is
-16.
```

The next 16 tables on the next 8 pages each reflect a state of the computer after an instruction is executed. We only show the steps till the value of fib[3] is computed. Most steps (state transitions) exhibit changes in two places: a control state change in PC and a data state change in a register or a memory location. We denote resulting data of these state changes in boldface. The lecturer can ask the students to continue drawing similar tables until the value of fib[4] is computed.

Note that these 16 tables show notations and memory layout simplified for ease of learning. The following table shows a realistic, more complex initial state on an x86 processor using the AT&T assembly language notations (Table 2.6).

Table 2.6 The same initial state on an x86 processor

CPU Content		Memory Content	
Register	Value	Address	Instruction
eflags		0x672	mov $0, %rbx
rip	0x672	0x679	mov %rbx, 0(%rax)
rax	0x201010	0x67c	mov $1, %rbx
rbx	0	0x683	mov %rbx, 8(%rax)
rsi	0	0x687	mov $2, %rsi
		0x68e<for_loop>	mov $0, %rbx
		0x695	add -16(%rax, %rsi, 8), %rbx
		0x69a	add -8(%rax, %rsi, 8), %rbx
		0x69f	mov %rbx, (%rax, %rsi, 8)
		0x6a3	inc %rsi
		0x6a6	cmp $50, %rsi
		0x6aa	jl 68e
		0x201010	
		0x201018	
		0x201020	
		0x201028	

Step 1: 0→R1. Also, PC←PC+1=1.

CPU Contents		Memory Contents	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**1**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**0**	3	MOV R1, M[R0+8]
R2		4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	
		20	
		28	
		36	

Step 2:R1→M[R0], i.e., 0→M[12]. Also, PC←PC+1=2.

CPU Content		Memory Content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**2**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	0	3	MOV R1, M[R0+8]
R2		4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	**0**
		20	
		28	
		36	

Step 3: 1→R1. Also, PC←PC+1=3.

CPU Content		Memory Content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**3**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**1**	3	MOV R1, M[R0+8]
R2		4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	
		28	
		36	

Step 4: R1→M[R0+8], i.e., 1→M[20]. Also, PC←PC+1=4.

CPU Content		Memory Content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**4**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	1	3	MOV R1, M[R0+8]
R2		4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	**1**
		28	
		36	

Step 5: 2→R2. Also, PC←PC+1=5.

CPU Content		Memory Content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**5**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	1	3	MOV R1, M[R0+8]
R2	**2**	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	
		36	

Step 6: 0→R1. Also, PC←PC+1=6.

CPU Content		Memory Content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**6**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**0**	3	MOV R1, M[R0+8]
R2	2	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	
		36	

Step 7: R1+M[R0+R2*8-16]→R1, i.e., 0+M[12]→R1. Also, PC←PC+1=7.

CPU Content		Memory Content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**7**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**0**	3	MOV R1, M[R0+8]
R2	2	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	
		36	

Step 8: R1+M[R0+R2*8-8]→R1, i.e., 0+M[20]→R1. Also, PC←PC+1=8.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**8**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**1**	3	MOV R1, M[R0+8]
R2	2	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	
		36	

Step 9: R1→M[R0+R2*8-0], i.e., 1→M[28]. Also, PC←PC+1=9.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**9**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	1	3	MOV R1, M[R0+8]
R2	2	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	**1**
		36	

Step 10: R2+1→R2. Also, PC←PC+1=10.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS		0	MOV 0, R1
PC	**10**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	1	3	MOV R1, M[R0+8]
R2	**3**	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	1
		36	

Step 11: Compare R2 to 51, result "<" →FLAGS. Also, PC←PC+1=11.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS	<	0	MOV 0, R1
PC	**11**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	1	3	MOV R1, M[R0+8]
R2	3	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	1
		36	

Step 12: If FLAGS is <, Loop→PC. Loop is 5, 5→PC.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS	<	0	MOV 0, R1
PC	**5**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	1	3	MOV R1, M[R0+8]
R2	3	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	1
		36	

Step 13: 0→R1. Also, PC←PC+1=6.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS	<	0	MOV 0, R1
PC	**6**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**0**	3	MOV R1, M[R0+8]
R2	3	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	1
		36	

Step 14: R1+M[R0+R2*8-16]→R1, i.e., 0+M[20]→R1. Also, PC←PC+1=7.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS	<	0	MOV 0, R1
PC	**7**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**1**	3	MOV R1, M[R0+8]
R2	3	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	1
		36	

Step 15: R1+M[R0+R2*8-8]→R1, i.e., 1+M[28]→R1. Also, PC←PC+1=8.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS	<	0	MOV 0, R1
PC	**8**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	**2**	3	MOV R1, M[R0+8]
R2	3	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	1
		36	

Step 16: R1→M[R0+R2*8-0], i.e., 1→M[36]. Also, PC←PC+1=9.

CPU content		Memory content	
Register	Value	Address	Instruction
FLAGS	<	0	MOV 0, R1
PC	**9**	1	MOV R1, M[R0]
R0	12	2	MOV 1, R1
R1	2	3	MOV R1, M[R0+8]
R2	3	4	MOV 2, R2
		5 Loop	MOV 0, R1
		6	ADD M[R0+R2*8-16], R1
		7	ADD M[R0+R2*8-8], R1
		8	MOV R1, M[R0+R2*8-0]
		9	INC R2
		10	CMP 51, R2
		11	JL Loop
		12	0
		20	1
		28	1
		36	**2**

2.4 Exercises

1. The binary representation of decimal number 14.875 is:
   (a) 1110.111
   (b) 1111.011
   (c) 1110.101
   (d) 1111.101
2. The binary representation of the two's complement of integer -12 is:
   (a) 00001100
   (b) 10001100
   (c) 01110100
   (d) 11110100
3. In Sect. 2.1, there is a question mark in the table about temperature in Beijing. The temperature in question has a value of 32 °C, which cannot be represented with only 5 bits. How to fix this problem?
   (a) Use 6 bits to represent temperature values from 0y°C to 63 °C.
   (b) Represent 32 °C and every other higher temperature by 11111, which is already used as the representation for 31 °C.

   (c) Represent 32 °C and every other higher temperature by 11111 and signals an overflow.
   (d) Use 5 bits in only those scenarios where the temperature values are constrained to the range from 0 °C to 31 °C.

4. Consider the design of a digital display for a thermometer. We need to convert analog temperature signals between −50 °C to 50 °C into digital display symbols. In other words, we need to be able to display all temperature readings: −50, −49 . . ., −01, 00, 01, . . ., 49, 50. How many bits are needed with each of the following three number representations? Put the correct capital letter in the parentheses of each line below.

(a) The unsigned integer format needs ()	X: 6 bits
(b) The simple signed integer format needs ()	Y: 7 bits
(c) The two's complement format needs ()	Z: Can not be done

5. Consider the following three number representations for eight-bit numbers. Put the correct capital letter in the parentheses of each line below.

(a) The smallest value of unsigned integer is ()	U: 00000000
(b) The largest value of unsigned integer is ()	V: 00000001
(c) The smallest value of simple signed integer is ()	W: 01111111
(d) The largest value of simple signed integer is ()	X: 10000000
(e) The smallest value of two's complement is ()	Y: 10000001
(f) The largest value of two's complement is ()	Z: 11111111

6. Consider the three number representations for eight-bit numbers. To show overflow conditions, put the correct capital letter in the parentheses of each line below.

(a) For unsigned integers, the result is smaller than ()	U: -128
(b) For unsigned integers, the result is larger than ()	V: -127
(c) For simple signed integers, the result is smaller than ()	W: 0
(d) For simple signed integers, the result is larger than ()	X: 127
(e) For two's complement integers, the result is smaller than ()	Y: 128
(f) For two's complement integers, the result is larger than ()	Z: 255

7. Refer to the algorithm for 8-bit integer adder in Sect. 2.1. Design an algorithm for a two's complement subtractor computing C=A-B, where A, B, C are 8-bit integers in two's complement representation. Verifying the correctness of the subtractor by putting the correct capital letter in the parentheses of each line below.

(a) When A=63 and B=64, the result of 63-64 is ()	V: 00000000
(b) When A=-63 and B=64, the result of (-63)-64 is ()	W: 00000001
(c) When A=64 and B=63, the result of 64-63 is ()	X: 01111111
(d) When A=64 and B=-63, the result of 63-(-64) is ()	Y: 10000001
(e) When A=-64 and B=-63, the result of (-64)-(-63) is ()	Z: 11111111

8. To represent ASCII characters, put the correct capital letter in the parentheses of each line below. Note that there are three types of number representations: binary, decimal, and hexadecimal.

(a) 00000000_2 is the ASCII encoding for the character ()	U: NUL
(b) $5A_{16}$ is the ASCII encoding for the character ()	V: SP
(c) 97_{10} is the ASCII encoding for the character ()	W: 0
(d) 0x20 is the ASCII encoding for the character ()	X: a
(e) 48_{10} is the ASCII encoding for the character ()	Y: Z
(f) 00101011_2 is the ASCII encoding for the character ()	Z: +

9. To display the question mark symbol, the correct statement is:

   (a) fmt.Printf("%c", '?')
   (b) fmt.Printf("%b", 63)
   (c) fmt.Printf("%c", 63)
   (d) fmt.Printf("%d", 63)
   (e) fmt.Printf("%c", ?)
   (f) fmt.Printf("%c", '63')

10. To print out the ASCII symbol for escape (ESC), the correct statement is:

   (a) fmt.Printf("%c", 'ESC')
   (b) fmt.Printf("%c", 00011011)
   (c) fmt.Printf("%c", 27)
   (d) fmt.Printf("%c", '27')

11. Regarding integer division and the mod operation, which of the following statements are/is correct?

   (a) $18 / 10 = 1.8$
   (b) $18 / 10 = 1$
   (c) $18 \% 10 = 8$
   (d) $18 \bmod 10 = 1$

12. The following program compares student name to a character string to see how many common characters there are.

```
package main
import "fmt"
func main() {
  var name string = "Alan Turing"
  var cs string = "Computer Science"
  sum := 0
  for i := 0; i < 11; i++ {
    for j := 0; j < len(cs); j++ {
      if name[i]==cs[j] {sum++}
    }
  }
  fmt.Printf("%d\n", sum)
}
```

The correct output is:

(a) 5
(b) 6
(c) 7
(d) 8
(e) 9

13. The following program does the same thing as Exercise 12. However, it follows good programming practice and is easier for human to understand.

```
package main
import "fmt"
const studentName   = "Alan Turing"
const targetString  = "Computer Science"
func main() {
  sum := 0
  for i := 0; i < len(studentName); i++ {
    for j := 0; j < len(targetString); j++ {
      if studentName[i]==targetString[j] {
        sum = sum + 1
      }
    }
  }
  fmt.Printf("%d\n", sum)
}
```

How has the new code improved over the code in Exercise 12?

(a) The two const statements use descriptive names studentName and targetString, instead of using non-descriptive name and cs.
(b) The two const statements use constant declaration, instead of variable declaration. Constant declaration is more appropriate since the two entities studentName and targetString do not change their values in the code.
(c) In the outer for loop, the expression i < len(studentName) gets rid of the magic number 11 in the expression i < 11 of the old code.
(d) Code of the main function does not depend on the specific values of studentName and targetString. We can compare a new student name, e.g., by changing "Alan Turing" to "Gordon Moore". The old code will fail.
(e) The new code has no improvement, because the code is longer.

14. Personalized programming exercise. Write a Go program to output the *student code* in the following way. Suppose Alan Turing's studentName "Alan Turing" and his studentNumber 8009970023 are given. Compute the sum of the ASCII encoding values of the eleven characters in the string "Alan Turing"; compute studentCode = studentNumber / sum / sum; then output the value of studentCode. For Alan Turing, the program outputs 7334.

Each student (e.g., Ada Smith) uses her/his student name and student number to generate the student code, with three constraints: (1) only format verb %c is used; (2) student number is a 10-digit decimal number; and (3) the program should follow good programming practice.

15. The von Neumann model of computer has the following features:

    (a) A computer consists of interconnected processor, memory and I/O devices.
    (b) Symbols are represented as binary digits (bits).
    (c) Data and programs are stored in memory.
    (d) A program is serially executed by executing one instruction after another.

16. Which of the following statements are/is correct for a typical computer?

    (a) The address of the instruction to be executed next is stored in the program counter (PC).
    (b) Every computer has an instruction set.
    (c) A program in execution has a Standard Input, a Standard Output, and a Standard Error devices.
    (d) A hard disk stores data. So, it is a memory device, not an I/O device.

17. Which of the following statements are/is correct regarding the *state* of a von Neumann computer?

    (a) The state of a computer refers to the contents of the registers.
    (b) The state of a computer refers to the contents of the memory.
    (c) The state of a computer refers to the contents of the I/O devices.
    (d) The state of a computer refers to the contents of the registers, the memory, and the I/O devices. However, this chapter focuses on the contents of the registers and the memory.

18. We want to use base, index and offset to find the byte address in memory of an array element a[i] of 64-bit integer. Given base=200 and index i=3, which of the following statements are/is correct?

    (a) The byte address of a[i] is 224, because address = base + index*8 + offset = 200 + 3*8 + 0 = 224.
    (b) The byte address of a[i] is 211, because address = base + index + offset = 200 + 3 + 8 = 211.
    (c) The byte address of a[i] is 267, because address = base + index + offset = 200 + 3 + 64 = 267.
    (d) The byte address of a[i] is 392, because address = base + index*64 + offset = 200 + 3*64 + 0 = 392.

19. Consider the loop body fib[i] = fib[i-1] + fib[i-2] in Fig. 2.8. Suppose the address of fib[i] is R0+R2*8. Which of the following statements are/is correct?

    (a) The address of fib[i-1] is R0+R2*8-8.
    (b) The address of fib[i-2] is R0+R2*8-8.
    (c) The address of fib[i-1] is R0+R2*8-16.
    (d) The address of fib[i-2] is R0+R2*8-16.

20. Refer to Table 2.5 and associated explanation text. Assume part of the initial computer state is shown in the following table.

CPU content					Memory content			
FLAGS	PC	R0	R1	R2	M[12]	M[20]	M[28]	M[[36]
<	0	12	6	3	2	1	2	3

How will the computer state change after executing each of the following instructions? Put the correct capital letter in the parentheses of each line below.

(a) MOV 0, R1 makes ()	U: FLAGS='<'
(b) MOV R1, M[R0+R2*8+8] makes ()	V: M[44]=6
(c) ADD M[R0+R2*8-16], R1 makes ()	W: PC=5
(d) INC R2 makes ()	X: R1=0
(e) CMP 51, R2 makes ()	Y: R1=7
(f) JL 5 makes ()	Z: R2=4

21. Refer to Table 2.5 and associated explanation text. Assume part of the initial computer state is shown in the following table.

CPU contents					Memory contents			
FLAGS	PC	R0	R1	R2	M[12]	M[20]	M[28]	M[[36]
<	0	12	6	3	2	1	2	3

How will the computer state change after executing each of the following instructions? Put the correct capital letter in the parentheses of each line below.

(a) MOV 0, R1 makes ()	U: PC=0
(b) MOV R1, M[R0+R2*8+8] makes ()	V: PC=1
(c) ADD M[R0+R2*8-16], R1 makes ()	W: PC=2
(d) INC R2 makes ()	X: PC=3
(e) CMP 51, R2 makes ()	Y: PC=4
(f) JL 5 makes ()	Z: PC=5

22. Digital symbols can be used to represent the following entities.

    (a) Numbers, such as integers, floating-point numbers, natural numbers.
    (b) Characters, such as ASCII symbols and Chinese characters.
    (c) Media contents, such as texts, picture, audio, video, books.
    (d) Processes of human endeavor, such as business processes, scientific processes, computational processes.

23. Fill out the following form of von Neumann model with data from your personal computer. Some example parameters of the lecturer's computer are shown in Table 2.4.

Processor		
Memory		
I/O devices	Storage	
	Keyboard	
	Display	
	Mouse	
	Network	

2.5 Bibliographic Notes

The chapter quotation is from Herbert Simon and Allen Newell [1], two pioneers of artificial intelligence. The term “von Neumann architecture” and its controversy can be found at [2, 3]. The website [4] offers an introductory tour of Go programming, with accessible hands-on examples.

References

1. Simon HA, Newell A (1976) Computer science as empirical inquiry: symbols and search. Commun ACM 19(3):11–126
2. Moye WT (1996) ENIAC: the Army-sponsored revolution. US Army Research Laboratory. ftp.arl.army.mil/mike/comphist/96summary/index.html
3. O’Regan G (2018) Von Neumann architecture. In: The innovation in computing companion. Springer, Cham, pp 257–259
4. https://tour.golang.org/welcome/1

Chapter 3
Logic Thinking

Computer science is the continuation of logic by other means.
—Georg Gottlob, 2009

Logic thinking is concerned with one of the key questions in computer science: what kind of problems can be *correctly* solved by computational processes? This question can be broken down into two issues:

- The **correctness** issue. How to rigorously define correctness of computational processes? In doing so, we can have a rigorous definition of computable problems: those problems for which there exist correct computational processes.
- The **generality** issue. Is there a computer that can be used to solve any computable problem?

We introduce two bit-accurate models of computation: Boolean logic and Turing machine, to rigorously define and analyze the correctness and generality of computational processes. Four main points are emphasized:

- With Boolean logic and Turing machine, we are able to accurately model a problem and the computational process to solve the problem. We can also define and verify the correctness of the solution.
- The above method is universal, that is, Boolean logic and Turing machine can be used to model all computable problems and their solutions.
- Boolean logic and Turing machines have limitations. There exist undecidable problems, paradoxes and incompleteness theorems.
- Logic thinking in computer science has differences from logic thinking of other sciences. Logic thinking in computer science emphasizes bit accuracy and automatic execution.

We proceed in the following sequence.

- To ensure the correctness of a computational process, we make sure that each step of the process is correct and that compositions of steps are correct.
- To ensure one step is correct, we use Boolean logic.
- To ensure the correctness of multiple steps, we use Turing machines.

Z. Xu, J. Zhang, *Computational Thinking: A Perspective on Computer Science*,
https://doi.org/10.1007/978-981-16-3848-0_3

- To see the power and limitation of Turing machines, we study Church-Turing Hypothesis and Gödel's incompleteness theorems.

3.1 Boolean Logic

Boolean logic is a formal logic system to reason about logic statements which can have true (1) or false (0) values. Boolean logic has manifested in propositional logic and predicate logic. It is a perfect match for computer science, because computers use binary values of 0 and 1 to represent digital symbols.

We first use three examples to illustrate what problems can be solved by Boolean logic. Solutions to these problems will be provided in later examples.

- The Congruent Triangles Problem, to show that Boolean logic can be used to solve mathematic problems, without using mathematic domain knowledge.
- The Impatient Guide Problem, to illustrate that problems in many application domains can be encoded as Boolean logic problems.
- The Adder Implementation Problem, to show that many computer hardware components can be implemented as Boolean logic expressions.

Example 3.1. The Congruent Triangles Problem
Let us consider a statement in geometry: *congruent triangles are also similar*. More precisely, if two triangles are congruent, they are also similar.

We are taught in geometry class that this statement is true. The teacher may even have shown us a proof, using geometry knowledge.

Now consider another statement, which is related to the original statement:

If two triangles are not similar, then they are not congruent.

Logic thinking can be used to show that the second statement holds. The proof is very simple. More importantly, it does not involve any knowledge in geometry.

Example 3.2. The Impatient Guide Problem
A tourist is traveling in the land of Oz and wants to go to the Emerald City. The tourist reaches a crossroad with paths P and Q, one of which leads to the Emerald City. There is a guide G at the crossroad, who comes from either the Honest Village or the Lying Village. Anyone from the Honest Village always tells the truth, and anyone from the Lying Village always tells lies. The guide is impatient, in that G only answers one question from the tourist, and the answer is either "Yes" or "No".

What question should the tourist ask the guide, to determine the correct path?

Example 3.3. The Adder Implementation Problem
In this example, we realize the addition operation of two n-bit numbers with Boolean logic. This book asks students to realize adders in several contexts, because adders are simple and fundamental. If we can do addition, we can also do many other operations.

3.1.1 Propositional Logic

Propositional logic is a logic system for reasoning about combinational propositions made of propositional variables and logic connectives. An example is the proposition "If the Earth is flat, then Alan Turing is a computer scientist". In this section, we will learn how to understand such a proposition and how to decide whether it is true or false.

3.1.1.1 Propositions and Logic Connectives

A **proposition** is a declarative sentence which has a truth-value, that is, being true or false. For example, "this book is written in English"; "Beijing is China's capital city". A proposition can contain one or more other propositions as parts. For example, "5 is a prime number *and* $5 \equiv 1 \pmod 4$".

We use variables x, y, z to denote proposition, and $x = 1$ means proposition x is true while $x = 0$ means proposition x is false.

The word "and" in the previous ample is a *logic connective* called *conjunction*. Propositional logic largely involves studying these kinds of logical connectives and the rules determining the truth-values of the propositions combined from simpler propositions and collectives. Five logical connectives are commonly used. Their definitions and illustrative examples are shown in Box 3.1.

Box 3.1. Commonly Used Logic Connectives

- **Conjunction** $\wedge$ (also called **AND**): $x \wedge y = 1$ if and only if $x = y = 1$.
  For example, the solution of $x^2 + 2x < 0$ is $x > -2$ and $x < 0$. We can describe it as $(x > -2) \wedge (x < 0)$.
- **Disjunction** $\vee$ (also called **OR**): $x \vee y = 0$ if and only if $x = y = 0$.
  For example, the solution of $x^2 + 2x \geq 0$ satisfies $x \leq -2$ or $x \geq 0$. We can describe it as $(x \leq -2) \vee (x \geq 0)$.
- **Negation** $\neg$ (also called **NOT**): $\neg x = 0$ if and only if $x = 1$.
  We also use $\bar{x}$ to represent the negation of x, that is, $\bar{x} = \neg x$.
- **Implication** $\rightarrow$: $(x \rightarrow y) = 1$ if and only if $x = 0$ or $y = 1$.
  We call x **premise** and y **conclusion**. Implication means $x \rightarrow y = 1$ if and only if the premise is false or the conclusion is true.
- **Exclusive-or** (also called **XOR**) $\bigoplus$: $x \bigoplus y = 1$ if and only if $x \neq y$.
  That is, $x \bigoplus y$ is true iff either x or y (but not both) is true. Note $x \bigoplus 1 = \neg x$, that is, we can use exclusive-or operation to realize negation.

Four of the five connectives are straightforward. The implication connective may look strange for beginners to logic. A key observation is that a false premise implies anything! Thus, both of the following propositions are true:

The Earth is flat $\rightarrow$ Alan Turing is a computer scientist
The Earth is flat $\rightarrow$ Alan Turing is not a computer scientist

Table 3.1 Truth table for conjunction, disjunction, implication, and exclusive-or

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

Table 3.2 Basic properties of propositional logic

Law	Logic equivalence	School logic (assume x=2, y=3, z=4)
Associativity	$(x \bullet y) \bullet z = x \bullet (y \bullet z)$,	$(2 \bullet 3) \bullet 4 = 2 \bullet (3 \bullet 4)$
	$(x + y) + z = x + (y + z)$	$(2 + 3) + 4 = 2 + (3 + 4)$
Commutativity	$x \bullet y = y \bullet x$	$2 \bullet 3 = 3 \bullet 2$
	$x + y = y + x$	$3 + 2 = 2 + 3$
Distributivity	$(x + y) \bullet z = (x \bullet z) + (y \bullet z)$	$(2 + 3) \bullet 4 = (2 \bullet 4) + (3 \bullet 4)$
	$(x \bullet y) + z = (x + z) \bullet (y + z)$	$(2 \bullet 3) + 4 \neq (2 + 4) \bullet (3 + 4)$
Identity	$x + 0 = x, x \bullet 1 = x$	$2 + 0 = 2, 2 \bullet 1 = 2$
Annihilator	$x \bullet 0 = 0, x + 1 = 1$	$2 \bullet 0 = 0, 2 + 1 \neq 1$
Idempotence	$x \bullet x = x, x + x = x$	$2 \bullet 2 \neq 2, 2 + 2 \neq 2$
Absorption	$(x \bullet y) + x = x, (x + y) \bullet x = x$	$(2 \bullet 3) + 2 \neq 2, (2 + 3) \bullet 2 \neq 2$
Complementation	$x + \neg x = 1, x \bullet \neg x = 0$	N/A

3.1.1.2 Truth Table

For any proposition, we can list its truth values on all of the possible combinations of values of the variables, to form its truth table. The following table lists the truth values for conjunction, disjunction, implication, and exclusive-or operations, given the combinations of the truth values of propositional variables x and y (Table 3.1).

3.1.1.3 Properties of Logic Connectives

We call a proposition without connectives a *primitive proposition*, and a proposition with one or more connectives a *combinational proposition*. The truth value of a primitive proposition is not decided by proposition logic, but by the environment or context where the primitive proposition is made. Proposition logic is concerned with the truth values of combinational propositions, given the truth values of their primitive propositions and the combinations with logic connectives.

From definitions of logic connectives in Box 3.1, we can derive basic properties of propositional logic, as listed in Table 3.2. Additional properties are listed in Table 3.3. We omit the proof and the reader can use truth table to verify these properties.

A simple way to learn these properties is to contrast them to the logic we learned from high-school math or algebra classes. Some basic properties of propositional logic in Table 3.2 do not hold in high-school logic, shown in red. Note that we use

Table 3.3 Additional properties of propositional logic

Law	Logic equivalence
De Morgan law	$\overline{x \wedge y} = \bar{x} \vee \bar{y}, \overline{x \vee y} = \bar{x} \wedge \bar{y}$
Implication defined in $\vee, \neg$	$x \rightarrow y = \bar{x} \vee y$
XOR defined in $\wedge, \vee, \neg$	$x \bigoplus y = (\bar{x} \wedge y) \vee (x \wedge \bar{y}), x \bigoplus y = (x \vee y) \wedge (\bar{x} \vee \bar{y})$
Associativity for XOR	$x \bigoplus (y \bigoplus z) = (x \bigoplus y) \bigoplus z$
Commutativity for XOR	$x \bigoplus y = y \bigoplus x$
Identity for XOR	$x \bigoplus 0 = x, x \bigoplus 1 = \bar{x}$

the more familiar addition symbol + and multiplication symbol • from school classes, to denote disjunction (∨) and conjunction (∧).

The associativity, commutativity, and identity laws hold for both propositional logic and high school mathematical logic. However, the distributivity and the annihilator laws only hold for multiplication but not addition. In high school math, multiplication distributes over addition, e.g., (2 + 3) • 4 = (2 • 4) + (3 • 4), but not addition distributed over multiplication, e.g., (2 • 3) + 4 ≠ (2 + 4) • (3 + 4). The other three laws of idempotence, absorption, and complementation do not hold at all in high school math.

Table 3.3 lists additional properties for propositional logic which mainly consider negation, implication, and exclusive-or operations.

3.1.1.4 Boolean Expression and Boolean Function

When we view logic connectives as operators, primitive and combinational propositions will become Boolean expressions. More formally, the set L of all **Boolean expressions** is recursively defined as follows.

1. Initially, let 0, 1, $x_1, \ldots, x_n \in L$, where $x_1, \ldots, x_n$ are primitive propositions for some natural number n. We also call 0 and 1 *Boolean constants*, and $x_1, \ldots, x_n$ *Boolean variables*.
2. Recursively apply negation, conjunction, disjunction, implication, and exclusive-or operations: If $x, y \in L$, then $\neg x$, $x \bullet y$, $x + y$, $x \rightarrow y$, $x \bigoplus y \in L$. Use the basic properties in Table 3.2 or 3.3 (sometimes called **Boolean algebra axioms**) to reduce all equivalent expressions into one expression.
3. Repeat Step 2 until L does not change any more.

We also have the notion of **Boolean functions**. An n-input-1-output Boolean function is a mathematical function $f: \{0, 1\}^n \rightarrow \{0, 1\}$, and the mapping is defined by the truth table of f. For instance, the equation $y = x_1 \bigoplus x_2 \bigoplus \cdots \bigoplus x_n$ defines a Boolean function to find the parity of $x_1, x_2, \cdots, x_n$, where the output is y and the n inputs are $x_1, x_2, \cdots, x_n$.

3.1.1.5 Normal Forms

Given any Boolean expression, we can use the basic properties of Tables 3.2 and 3.3 to find an equivalent expression which contains only AND, OR, and NOT. For example, $(x \vee y) \to z = \left(\overline{x \vee y}\right) \vee z = (\overline{x} \wedge \overline{y}) \vee z = (\overline{x} \vee z) \wedge (\overline{y} \vee z)$. Note that there are several different ways to present an expression with only AND, OR, and NOT. For example, the above example provides three different ways to represent $(x \vee y) \to z$ with only AND, OR, and NOT. However, there is a uniform way to achieve this via the truth table. Let us write down the truth table for $(x \vee y) \to z$ (Table 3.4).

From the truth table, we know that there are three cases where the expression $(x \vee y) \to z$ is false: (1) $x = 0,\ y = 1,\ z = 0$; (2) $x = 1, y = z = 0$; (3) $x = y = 1,\ z = 0$. For the remaining five cases, the expression is true. For each case when the expression is true, we can use a product clause to represent it. For example, clause $\overline{x} \wedge \overline{y} \wedge \overline{z}$ represents the case $x = y = z = 0$, since $\overline{x} \wedge \overline{y} \wedge \overline{z} = 1$ if and only if $x = y = z = 0$. We then use $\vee$ to connect those true clauses to represent the whole expression:

$$(\overline{x} \wedge \overline{y} \wedge \overline{z}) \vee (\overline{x} \wedge \overline{y} \wedge z) \vee (\overline{x} \wedge y \wedge z) \vee (x \wedge \overline{y} \wedge z) \vee (x \wedge y \wedge z).$$

It is easy to check that the truth table for the above proposition is the same as the truth table for $(x \vee y) \to z$. Thus, they are equivalent propositions. We call it the **disjunctive normal form**. Actually, for any proposition, we can use the above method to write down its disjunctive normal form.

Theorem: For any proposition $F(x_1, x_2, \ldots, x_n) \not\equiv 0$ with n variables, we can uniquely represent it as the following disjunctive normal form:

$$\boldsymbol{F(x_1, x_2, \ldots, x_n) = Q_1 \vee Q_2 \vee \cdots \vee Q_m}$$

where each product $Q_i = l_1 \wedge l_2 \wedge \cdots \wedge l_n$, and $l_j = x_j$ or $\overline{x_j}$.

This theorem is obtained by looking at 1-valued rows in a truth table. If we look at 0-valued rows in the truth table, can we also write an expression for $(x \vee y) \to z$? The answer is YES. For each 0-valued row, we can write the representations connected

Table 3.4 Truth table for proposition $(x \vee y) \to z$

x	y	z	$(x \vee y) \to z$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

by OR. For example, the row "$x = 0,\ y = 1,\ z = 0$" can be represented by $(\mathrm{x} \vee \overline{\mathrm{y}} \vee \mathrm{z})$. It means $(\mathrm{x} \vee \overline{\mathrm{y}} \vee \mathrm{z}) = 0$ if and only if $x = 0,\ y = 1,\ z = 0$. Finally, we use AND to connect the representations for the three 0-valued rows.

$$(x \vee y) \rightarrow z = (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee y \vee z) \wedge (\overline{x} \vee \overline{y} \vee z).$$

This is another way to write an equivalent proposition for any proposition, and we call it the **conjunctive normal form**.

Theorem: For any proposition $F(x_1, x_2, \ldots, x_n) \not\equiv 1$ with n variables, we can uniquely represent it as the following conjunctive normal form:

$$\boldsymbol{F(x_1, x_2, \ldots, x_n) = Q_1 \wedge Q_2 \wedge \cdots \wedge Q_m}$$

where each sum $Q_i = l_1 \vee l_2 \vee \cdots \vee l_n$, and $l_j = x_j$ or $\overline{x_j}$.

3.1.1.6 The Number of Boolean functions

Given a natural number $n > 0$, how many different Boolean functions are there with n input variables?

Note that a Boolean function may be represented as two or more equivalent Boolean expressions. Two Boolean expressions are equivalent if they have the same truth table. This is the same as saying that two Boolean expressions are equivalent if they have the same disjunctive normal form or if they have the same conjunctive normal form. Two Boolean functions are different, if they do not have equivalent Boolean expressions. Thus, all equivalent Boolean expressions are counted as one, when computing the number of different Boolean functions.

The original question can be reduced to the question: how many different truth tables there are for n input variables and one output variable. Let us look at a truth table of n input variables in general.

x_1	x_2	...	x_{n-1}	x_n	y
0	0	...	0	0	0 or 1
0	0	...	0	1	0 or 1
0	0	...	1	0	0 or 1
0	0	...	1	1	0 or 1
...	...	...	...	...	0 or 1
1	1	...	1	0	0 or 1
1	1	...	1	1	0 or 1

Any truth table has 2^n rows. Consequently, the y column has 2^n cells, and each can have a 0 or 1 value. Each different configuration of 0/1 values in the 2^n cells represents a different truth table. There are 2^{2^n} configurations. Thus, there are 2^{2^n} truth tables. So, there are 2^{2^n} distinct Boolean functions.

The above result implies that any given Boolean function can be implemented by a Boolean expression. From the normal form theorems, any Boolean function can be implemented by AND, OR, NOT operations. This gives an affirmative answer to the Adder Implementation Problem, since Adder is a Boolean function.

Example 3.4. The Numbers of Boolean Functions of One and Two Variables
For any given integer $n > 0$, there are 2^{2^n} distinct Boolean functions. Let us understand this result more concretely by explicitly enumerating all Boolean expressions for n=1 and n=2. In this example, we only apply negation, conjunction and disjunction operations in each round.

First consider the case when $n = 1$. The number of Boolean functions is 2^{2^1}=4. We can use the recursive definition of Boolean expressions to find all Boolean functions of one input variable x. The four functions shown below with their truth tables. They are: y is always false, y is always true, $y = x$, and y = NOT x.

x	y
0	**0**
1	**0**

$y = 0$

x	y
0	**1**
1	**1**

$y = 1$

x	y
0	**0**
1	**1**

$y = x$

x	y
0	**1**
1	**0**

$y = \overline{x}$

Now consider the case when $n = 2$. The number of Boolean functions is 2^{2^2}=16. Again, we use the recursive definition of Boolean expressions to find all Boolean functions of two input variables x_1 and x_2.

Round 1. $y = 0, y = 1, y = x_1, y = x_2, y = \overline{x_1}, y = \overline{x_2}$. Note that we simplify the process by putting negations of Boolean variables in the initial step. At the end of round 1, we have the Boolean expression set $L = \{0, 1, x_1, x_2, \overline{x_1}, \overline{x_2}\}$. The corresponding truth tables are shown below.

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**0**

$y = 0,$

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**1**

$y = 1,$

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**1**

$y = x_1,$

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**1**

$y = x_2$

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**0**
1	1	**0**

$y = \overline{x_1}$,

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**1**
1	1	**0**

$y = \overline{x_2}$

Round 2. Apply AND, OR, NOT to L, and use the axioms of Boolean expressions to eliminate redundant expressions. We add to L eight new expressions:
$\bar{x_1} \vee \bar{x_2}$, $\bar{x_1} \vee x_2$, $x_1 \vee \bar{x_2}$, $x_1 \vee x_2$, $\bar{x_1} \wedge \bar{x_2}$, $\bar{x_1} \wedge x_2, x_1 \wedge \bar{x_2}$, $x_1 \wedge x_2$
We have the new Boolean expression set

$$L = \{0, 1, x_1, x_2, \bar{x_1}, \bar{x_2}; \bar{x_1} \vee \bar{x_2}, \bar{x_1} \vee x_2, x_1 \vee \bar{x_2}, x_1 \vee x_2,\ \bar{x_1} \wedge \bar{x_2}, \bar{x_1} \wedge x_2, x_1 \wedge \bar{x_2}, x_1 \wedge x_2\}.$$

The corresponding truth tables for the eight new expressions are shown below.

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

$y = \overline{x_1} \vee \overline{x_2}$,

x_1	x_2	y
0	0	**1**
0	1	**1**
1	0	**0**
1	1	**1**

$y = \overline{x_1} \vee x_2$,

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**1**
1	1	**1**

$y = x_1 \vee \overline{x_2}$,

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

$y = x_1 \vee x_2$,

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

$y = \overline{x_1} \wedge \overline{x_2}$,

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**0**

$y = \overline{x_1} \wedge x_2$,

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**0**

$y = x_1 \wedge \overline{x_2}$,

x_1	x_2	y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

$y = x_1 \wedge x_2$

Round 3. Apply AND, OR, NOT to L, and use the axioms of Boolean expressions to eliminate redundant expressions. We add to L two new expressions:

$$(\bar{x_1} \wedge \bar{x_2}) \vee (x_1 \wedge x_2),\ \ (\bar{x_1} \wedge x_2) \vee (x_1 \wedge \bar{x_2})$$

We have the new Boolean expression set

$$L = \{0, 1, x_1, x_2, \bar{x_1}, \bar{x_2}; \bar{x_1} \vee \bar{x_2}, \bar{x_1} \vee x_2, x_1 \vee \bar{x_2}, x_1 \vee x_2,\ \bar{x_1} \wedge \bar{x_2}, \bar{x_1} \wedge x_2, x_1 \wedge \bar{x_2}, x_1 \wedge x_2; (\bar{x_1} \wedge \bar{x_2}) \vee (x_1 \wedge x_2), (\bar{x_1} \wedge x_2) \vee (x_1 \wedge \bar{x_2})\}.$$

The corresponding truth tables for the two new expressions are shown below.

x_1	x_2	y
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

$$y = (\overline{x_1} \wedge \overline{x_2}) \vee (x_1 \wedge x_2), \quad y = (\overline{x_1} \wedge x_2) \vee (x_1 \wedge \overline{x_2})$$

Round 4. Apply AND, OR, NOT to L. We get no more new expressions. Stop. The final set of all 16 Boolean expressions are in the L obtained in Round 3.

Example 3.5. The Adder Implementation Problem, Revisited
Students are asked to implement an adder, which takes two n-bit numbers X and Y as inputs and produces an n-bit number Z as the output. The adder is an n-input-n-output Boolean function. Since any n-input-1-output Boolean function can be implemented by an n-variable Boolean expression, we can theoretically use n such Boolean expressions to implement an n-input-n-output Boolean function. In practice, we can often have better implementations.

Let us start at n=1. A **full adder** has three input variables x_1, y_1, c_0 and two output variables z_1, c_1. Sometimes we simplify the variables as x, y, c_{in}, z, c_{out}, where $c_{in} = c_0$ is the carry-in, and $c_{out} = c_1$ is the carry-out. The function of a full adder can be understood as $x_1 + y_1 + c_{in} = (c_{out}z)_2$ where x_1, y_1, c_{in} are three 1-bit binary numbers and $(c_{out}z)_2$ is a 2-bit binary number. We can use 2 Boolean expressions to compute z, c_{out}. See the following equations and truth table:

$$z = x \bigoplus y \bigoplus c_{in}; \quad c_{out} = (x \wedge y) \vee \left(\left(x \bigoplus y\right) \wedge c_{in}\right)$$

c_{in}	x	y	z	c_{out}
0	0	0	**0**	**0**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

Students are asked to verify that these equations correctly implement the addition of 1-bit binary numbers (unsigned integers).

With the full adder concept in place, it is straightforward to implement an n-bit adder, by cascading n full adders, where the carry-out of the current bit serves as the carry-in of the next bit. The equations relating the input variables to output variables follow:

$$\begin{aligned}
&z_1 = x_1 \bigoplus y_1 \bigoplus c_0; \quad c_1 = (x_1 \wedge y_1) \vee ((x_1 \bigoplus y_1) \wedge c_0) \\
&z_2 = x_2 \bigoplus y_2 \bigoplus c_1; \quad c_2 = (x_2 \wedge y_2) \vee ((x_2 \bigoplus y_2) \wedge c_1) \\
&z_3 = x_3 \bigoplus y_3 \bigoplus c_2; \quad c_3 = (x_3 \wedge y_3) \vee ((x_3 \bigoplus y_3) \wedge c_2) \\
&\qquad\qquad\qquad \cdots\cdots\cdots\cdots \\
&z_{n-1} = x_{n-1} \bigoplus y_{n-1} \bigoplus c_{n-2}; \quad c_{n-1} = (x_{n-1} \wedge y_{n-1}) \vee ((x_{n-1} \bigoplus y_{n-1}) \wedge c_{n-2}) \\
&z_n = x_n \bigoplus y_n \bigoplus c_{n-1}; \quad c_n = (x_n \wedge y_n) \vee ((x_n \bigoplus y_n) \wedge c_{n-1})
\end{aligned}$$

The above equations realize the addition of n-bit binary numbers X and Y: $(x_n...x_1)_2 + (y_n...y_1)_2 + c_0 = (c_n z_n...z_1)_2$.

3.1.1.7 (***) Kleene Logic

In Table 3.2, we show that when logic connectives AND and OR are used as multiplication and addition operators, respectively, we have a Boolean algebra, the Boolean logic of which is different from the familiar logic of school algebra.

We also have shown that given any Boolean function, there exists a Boolean expression that implements the Boolean function. In other words, any Boolean function can be implemented by AND, OR, NOT operators over Boolean constants and Boolean variables. This beautiful property should not be taken for granted. A slight change could nullify this property. Let us look at an example.

The set of all **Kleene expressions** L is recursively defined as follows.

1. Initially, let 0, 1, $x_1, \ldots, x_n \in L$, where 0 and 1 are *Kleene constants*, and $x_1, \ldots, x_n$ are *Kleene variables*.
2. Recursively apply NOT, AND, OR to L: If $x, y \in L$, then $\neg x$, $x \bullet y$, $x+y \in L$. Use the basic properties in Table 3.5 (called **Kleene algebra axioms**) to reduce all equivalent expressions into one expression, that is, to eliminate redundancy.
3. Repeat Step 2 until L no longer changes.

Note that the Complementation law $x + \neg x = 1$ does not hold anymore. It is replaced by three weaker laws: the de Morgan law, the double negation law, and the product law. This seemingly slight change to Boolean logic makes the following

Table 3.5 Contrasting the axioms of Boolean algebra and Kleene algebra

Law	Boolean algebra	Kleene algebra
Associativity	$(x \bullet y) \bullet z = x \bullet (y \bullet z)$,	$(x \bullet y) \bullet z = x \bullet (y \bullet z)$,
	$(x + y) + z = x + (y + z)$	$(x + y) + z = x + (y + z)$
Commutativity	$x \bullet y = y \bullet x$	$x \bullet y = y \bullet x$
	$x + y = y + x$	$x + y = y + x$
Distributivity	$(x + y) \bullet z = (x \bullet z) + (y \bullet z)$	$(x + y) \bullet z = (x \bullet z) + (y \bullet z)$
	$(x \bullet y) + z = (x + z) \bullet (y + z)$	$(x \bullet y) + z = (x + z) \bullet (y + z)$
Identity	$x + 0 = x, x \bullet 1 = x$	$x + 0 = x, x \bullet 1 = x$
Annihilator	$x \bullet 0 = 0, x + 1 = 1$	$x \bullet 0 = 0, x + 1 = 1$
Idempotence	$x \bullet x = x, x + x = x$	$x \bullet x = x, x + x = x$
Absorption	$(x \bullet y)+x = x,\ (x+y)\bullet x = x$	$(x \bullet y)+x = x,\ (x+y)\bullet x = x$
Complementation	$x + \neg x = 1, x \bullet \neg x = 0$	N/A
de Morgan		$\neg(x+y)= \neg x \bullet \neg y,\ \neg(x \bullet y)= \neg x + \neg y$
Double Negation		$\neg\neg x = x$
Product		$p\bullet x_i+p\bullet\neg x_i = p+p\bullet x_i+p\bullet\neg x_i$

Table 3.6 The numbers of distinct Boolean and Kleene expressions of n variables

n	# of Distinct Boolean expressions	# of Distinct Kleene expressions
1	$2^{2^1} = 4$	6
2	$2^{2^2} = 16$	84
3	$2^{2^3} = 256$	43918
4	$2^{2^4} = 65536$	160297985276
In general	2^{2^n}	Unknown but $< 2^{3^n}$

statement to be false: any Kleene function can be implemented by AND, OR, NOT operators over Kleene constants and Kleene variables.

In Boolean algebra, $p\bullet x_i+p\bullet\neg x_i = p\bullet(x_i+\neg x_i)= p$. But in Kleene algebra, this no longer holds. We have $p\bullet x_i+p\bullet\neg x_i = p+p\bullet x_i+p\bullet\neg x_i$, where p is a product of Kleene variables and their negations. For instance, given the product $p = \neg x_1\bullet x_2$, from the product law we have $(\neg x_1\bullet x_2)\bullet x_3+(\neg x_1\bullet x_2)\bullet\neg x_3 = \neg x_1\bullet x_2 + \neg x_1\bullet x_2\bullet x_3 + \neg x_1\bullet x_2\bullet\neg x_3$, not $(\neg x_1\bullet x_2)\bullet x_3+(\neg x_1\bullet x_2)\bullet\neg x_3 = \neg x_1\bullet x_2$.

We know that there are 2^{2^n} distinct Boolean expressions of n variables, with the assumption that equivalent expressions are counted as one expression.

How many distinct Kleene expressions of n variables are there, for a given $n>0$? This is still an open problem. We compare the numbers of distinct Boolean and Kleene expressions of n variables in Table 3.6. Note that we still do not know the formula for the number of distinct Kleene expressions, but do know that this number is less than 2^{3^n}.

To have a more concrete understanding, let us enumerate all Kleene expressions of one variable x. Initially, the set L of Kleene expressions is $L = \{0, 1, x\}$.

Round 1. We obtain a new expression $\bar{x}$.The new $L = \{0, 1,\ \ x;\ \ \bar{x}\}$.

Round 2. For Boolean expressions, we would stop here, as L no longer changes.

Table 3.7 Truth table showing the truth values of AND, OR, NOT operators in Kleene logic

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	**0**	**0**	**1**
0	I	**0**	**I**	**1**
0	1	**0**	**1**	**1**
I	0	**0**	**I**	**I**
I	I	**I**	**I**	**I**
I	1	**I**	**1**	**I**
1	0	**0**	**1**	**0**
1	I	**I**	**1**	**0**
1	1	**1**	**1**	**0**

Round 2. For Kleene expressions, we continue and obtain two new expressions $x \cdot \bar{x}, x + \bar{x}$, the new $L = \{0, 1,\ \ x;\ \ \bar{x};\ \ x \cdot \bar{x}, x + \bar{x}\}$.

Round 3. Stop, since L no longer changes.

Note that the number of Boolean expressions of one variable x is 2^{2^1}=4. These four expressions are: always false, always true, identical to x, and NOT x. For Kleene logic, the number of Kleene expressions is 6. The two new expressions are: x AND (NOT x), and x OR (NOT x), which are absent from Boolean logic.

We also have the notion of Kleene functions, similar to Boolean functions. An n-input-1-output **Kleene function** is a mathematical function

f: $\{0, \mathrm{I}, 1\}^n \rightarrow \{0, \mathrm{I}, 1\}$, and the mapping is defined by the truth table of f. Different from binary Boolean logic, Kleene logic is ternary, in that we have three values. Besides 0 (False) and 1 (True), we have a new middle value I for Indeterminate. Table 3.7 shows the truth values of the AND, OR, NOT operators on two variables x and y.

Note that there are 3^{3^n} n-input-1-output distinct Kleene functions and truth tables. For n=1, there are $3^{3^1} = 27$ distinct functions but only six distinct Kleene expressions. Many Kleene functions cannot be represented by Kleene expressions. Thus, in Kleene logic, some Kleene functions cannot be implemented by AND, OR, NOT.

3.1.1.8 Using Propositional Logic to Solve Problems

We discuss four examples to show how logic helps produce correct computational processes.

Example 3.6. The Congruent Triangles Problem, Revisited

Given any conditional statement "if P then Q", we denote it in propositional logic as P→Q. The negation of this statement is NOT(if P then Q), or ¬(P→Q). The two statements are related. Only one is true.

Using the triangles example, let us call "P→Q" the original statement, where P stands for "two triangles are congruent" and Q stands for "two triangles are similar". Four types of statements are derived from the original statement, as shown in Table 3.8. These four types of statements are logically related.

Table 3.8 The Converse, Inverse, Contrapositive, and Negation of a conditional statement

Statement type	Logic form	Triangles example
Original	$P \to Q$	If two triangles are congruent, then they are similar
Converse	$Q \to P$	If two triangles are similar, then they are congruent
Inverse	$(\neg P) \to (\neg Q)$	If two triangles are not congruent, then they are not similar
Contrapositive	$(\neg Q) \to (\neg P)$	If two triangles are not similar, then they are not congruent
Negation	$\neg(P \to Q)$	NOT (If two triangles are congruent, then they are similar)

These logic relationships can be used to arrive at new statements and their truth values, sometimes without needing domain knowledge of geometry.

The original statement, "if two triangles are congruent, then they are similar", is a true statement. We know this from geometry. Congruent triangles have the same shape and size. Similar triangles have the same shape. Two triangles, having the same shape and size, of course have the same shape. Thus, they are similar. That is, $P \to Q$ is a true proposition.

The **converse** of the original statement is "if two triangles are similar, then they are congruent". The converse of the conditional statement $P \to Q$ is the statement obtained by exchanging the position of P and Q, namely, $Q \to P$. From geometry knowledge, we know this statement is false. Two similar triangles can have the same shape but different sizes, thus are not congruent. That is, $Q \to P$ is a false proposition.

The **inverse** of the original statement is "if two triangles are not congruent, then they are not similar". The inverse of the conditional statement $P \to Q$ is the statement obtained by negating both P and Q, namely, $(\neg P) \to (\neg Q)$.

Now, what is the truth value of the inverse statement? Is it true or false? We can obtain the answer without knowing geometry. In fact, we know immediately that the inverse statement $(\neg P) \to (\neg Q)$ in this example is a false proposition, because (1) the converse statement is false, and (2) *the converse and the inverse statements are logically equivalent*. The second point can be proven easily by showing that $(\neg P) \to (\neg Q) = Q \to P$.

$(\neg P) \to (\neg Q)$ The given inverse statement
$= \neg(\neg P) \vee (\neg Q)$ by implication property $P \to Q = \neg P \vee Q$
$= P \vee (\neg Q)$ eliminate double negation
$= \neg Q \vee P$ use communicative law
$= Q \to P$ obtain the converse by implication property.

The **contrapositive** of the original statement is "if two triangles are not similar, then they are not congruent". The contrapositive of the conditional statement $P \to Q$ is the statement obtained by negating both P and Q, and then exchanging positions, namely, $(\neg Q) \to (\neg P)$. We can show that *the contrapositive statement and the original statement are logically equivalent*. That is, $(\neg Q) \to (\neg P)$ is equivalent to $P \to Q$. The proof is the same as showing $(\neg P) \to (\neg Q) = P \to Q$.

Finally, the **negation** of the original statement is "NOT (If two triangles are congruent, then they are similar)". The negation of the conditional statement $P \to Q$ is just the logic negation, namely, $\neg(P \to Q)$. It can be transformed into other equivalent forms: $\neg(P\rightarrow Q) = \neg(\neg P \vee Q) = (\neg\neg P) \wedge (\neg Q) = P \wedge (\neg Q)$. That is, "Two triangles are congruent AND they are not similar", which is a false statement.

Note that the negation is not the same as the inverse. The original statement AND its negation always form a contradiction: $(P\rightarrow Q) \wedge \neg(P\rightarrow Q) = \text{FALSE}$. The original statement AND its inverse yield "P is identical to Q": $(P\rightarrow Q) \wedge (\neg P\rightarrow\neg Q) = P\equiv Q$.

Example 3.7. The Impatient Guide Problem, Revisited
A tourist is traveling in the land of Oz and wants to go to the Emerald City. The tourist reaches a crossroad with paths P and Q, one of which leads to the Emerald City. There is a guide G at the crossroad, who comes from either the Honest Village or the Lying Village. Anyone from the Honest Village always tells the truth, and anyone from the Lying Village always tells lies. The guide is impatient, in that G only answers one question from the tourist, and the answer is either "Yes" or "No".

What question should the tourist ask the guide, to determine the correct path?

The main difficulty is as follows. On one hand, the tourist needs to collect information which apparently needs at least two answers to Yes-No questions. On the other hand, the impatient guide only answers one question. Fortunately, propositional logic tells us that we can use proper connectives to combine two propositions into a single proposition. One of the propositions should contain information about the Honest or Lying Village, the other should be about the path to the Emerald City. With this line of thought, we come up with the following question:

> Are your answers the same, to the two questions "are you from the Honest Village" and "does path P lead to the Emerald City"?

If the answer is "Yes", take path P; if the answer is "No", take path Q.

To make the above reasoning clearer, let us use propositional notations to denote the solution.

- H denotes the proposition "G is from the Honest Village". That is, H=1 means G is from the Honest Village; H=0 means G is from the Lying Village.
- S denotes the proposition "Path P leads to the Emerald City". That is, S=1 means path P leads to the Emerald City; S=0 means path Q leads to the Emerald City.

With these notations, the single question to ask is $\neg(H\bigoplus S)=?$. However, the answer we get is not the true value of $\neg(H\bigoplus S)$ since it depends on whether the guide G comes from Honest Village or not. If G is from the Honest Village, we will get the true value of $\neg(H\bigoplus S)$; while if G is from the Lying Village, we will get the true value of $H\bigoplus S$. If we use the propositional notations to represent the above argument, the answer we will hear is actually $(\neg H)\bigoplus \neg(H\bigoplus S)$. By applying the properties in Table 3.2 and 3.3 or calculating the truth table of this Boolean expression, it is easy to find that $(\neg H)\bigoplus \neg(H\bigoplus S) = S$. Thus, we should choose path P if the answer we get is "Yes" and choose path Q if the answer we get is "No".

To make the argument clearer, we verify the correctness of the question and its answer by using a truth table.

H	S	$\neg(H \oplus S)$	Comments
0	0	1	G is lying and the true value of the question is "Yes"
			The answer is "No", take path Q
0	1	0	G is lying and the true value of the question is "No"
			The answer is "Yes", take path P
1	0	0	G is telling the truth and the true value of the question is "No"
			The answer is "No", take path Q
1	1	1	G is telling the truth and the true value of the question is "Yes"
			The answer is "Yes", take path P

Example 3.8. The Parity Program to Show Logic and Bit-Shift Operations
The **parity** of a number refers to whether the number's bits have an even number of 1's (parity is 0) or an odd number of 1's (parity is 1). The program parity.go below computes the parity values of 63 and 127. The parity function computes parityValue $= X_0 \oplus X_1 \oplus X_2 \oplus \ldots \oplus X_{63}$ for any 64-bit integer $X = (X_{63}X_{62}\ldots X_0)_2$, where $\oplus$ is the XOR operator. So, 63 = 0...00111111 has six 1's (even, parity is 0), and 127 = 0...01111111 has seven 1's (odd, parity is 1).

The statement

```
parityValue ^= X & 1
```

is a shorthand for

```
parityValue = parityValue ^ (X & 1)
```

where & is the bitwise AND operator and ^ is the bitwise XOR operator. More specifically, let $X = (X_{63}X_{62}\ldots X_0)_2$ and $Y = (Y_{63}X_{62}\ldots Y_0)_2$, we have $X \;\&\; Y = (X_{63} \wedge Y_{63}, X_{62} \wedge Y_{62}, \ldots X_0 \wedge Y_0)_2$ and $X \;\hat{}\; Y = (X_{63} \oplus Y_{63}, X_{62} \oplus Y_{62}, \ldots X_0 \oplus Y_0)_2$.The expression (X & 1) clears all bits of X but keeps the rightmost bit intact, that is, $X \;\&\; 1 = (00\ldots00X_0)_2$. The expression parityValue ^ (X & 1) computes (the current parityValue) $\oplus$ (last bit of X). Finally, the statement X = X >> 1 right shifts X one bit, for the next iteration. That is, X >> 1= $(0X_{63}X_{62}\ldots X_2X_1)$ (Fig. 3.1).

Example 3.9. Program to Hide a Character in a Byte Array
In the Text Hider project, students are asked to hide a text file in an image file. This example does a much simpler task of hiding an ASCII character 'K' in a byte array A=[11010001, 11001001, 11011010, 11011010] = [D1, C9, DA, DA]. The program replace.go in Fig. 3.2 does this by replacing the least significant two bits of the four elements of array A, with the eight bits of character 'K'. Every element A[i] hides two bits of 'K', as the following table shows (Table 3.9).

```
package main
import "fmt"
func parity(X int) int {
  parityValue := 0
  for i := 0; i<64; i++ {          // X is a 64-bit integer
    parityValue ^= X & 1           // parityValue = parityValue ^ (X & 1)
    X = X >> 1                     // shift X right one bit
  }
  return parityValue
}
func main() {
  a := 63                          // 63 = 00111111 has six 1's
  fmt.Println(parity(a))
  a = 127                          // 127 = 01111111 has seven 1's
  fmt.Println(parity(a))
}
```

(a) Source code of program parity.go

```
> go run parity.go
0
1
>
```

(b) Running parity.go to produce the output

Fig. 3.1 Using parity.go to illustrate logic and shift operations

The program uses tab \t to align the two lines of printing outputs, to better see the changes made to the last two bits of the array elements. The main work is done in the for loop, which iterates over the four elements A[0] to A[3], with the index values changing from i=0, 1, 2, to 3. We only need to look at the detailed case when i=0. The other cases are similar. When i=0, we have data='K'=01001011 and A[i]=A [0]= 11010001. The results of the loop body are shown below step-by-step.

```
 v := data & 0x3       v=   01001011 & 00000011 = 00000011
                       // retain rightmost 2 bits of 'K'
 A[i] = A[i] & 0xFC     A[i]=   11010001 & 11111100 = 11010000
                       // clear rightmost 2 bits of A[i]
 A[i] = A[i] | v       A[i]=  11010000 | 00000011 = 11010011
                       // set last 2 bits of A[i] with those of 'K'
data = data >> 2       data=  00010010
                       // shift 'K' 2 bits to the right
```

```
package main
import "fmt"
func main() {
  A := [4]byte{0xD1,0xC9,0xDA,0xDA}
  fmt.Printf("Before: \tA = [%b %b %b %b]\n",A[0],A[1],A[2],A[3])
  data := byte('K')
  for i := 0; i < len(A); i++ {
    v := data & 0x3                // retain last 2 bits of 'K'
    A[i] = A[i] & 0xFC             // clear last 2 bits of A[i]
    A[i] = A[i] | v                // set last 2 bits of A[i] with those of 'K'
    data = data >> 2               // repeat with the next 2 bits of 'K'
  }
  fmt.Printf("After: \t\tA = [%b %b %b %b]\n",A[0],A[1],A[2],A[3])
}
```

(a) Source code of program replace.go.

```
> go run replace.go
Before:         A = [11010001 11001001 11011010 11011010]
After:          A = [11010011 11001010 11011000 11011001]
>
```

(b) Running replace.go to produce the output

Fig. 3.2 Using replace.go to illustrate logic and shift operations

Table 3.9 Values of elements of array A before and after replacing the least significant two bits with character ‘K’ = 75 = 01001011

Array element	Before	After
A[0]	110100**01**	110100**11**
A[1]	110010**01**	110010**10**
A[2]	110110**10**	110110**00**
A[3]	110110**10**	110110**01**

3.1.2 Predicative Logic

Predicative logic is also called first-order logic. It contains propositional logic as well as predicates and quantifiers. Predicative logic has more expressive power than propositional logic. That is, predicative logic can be used to rigorously express some logic statements which ppositional logic cannot do.

3.1.2.1 Predicate and Quantifier

A **predicate** can be viewed as a proposition with one or more input variables. A predicate takes an entity or entities as input variables to produce an output of either True or False value. For instance, consider the two sentences “Socrates is a philosopher” and “Plato is a philosopher”. In propositional logic, these sentences are viewed as being unrelated and may be denoted, for example, by propositional variables p and q. However, in predicative logic, we can use predicate Phil(x) to represent "x. is a philosopher", where x is an input variable. Thus, if "a" represents "Socrates", then Phil(a) means "Socrates is a philosopher". Phil(x) is a predicate with one input variable x. Phil(a) is a predicate with the input variable x instantiated with the entity constant a.

There are two **quantifiers** in predicative logic. The universal quantifier $\forall$ means "for all", "for any", "for every". The existential quantifier $\exists$ means "there exists". For instance, we can have the following statements and their expressions.

All philosophers are mortals.	$\forall x$ [Phil(x)→Mortal(x)].
Socrates is a philosopher.	Phil(a).
Socrates is a mortal.	Mortal(a).
There exists a philosopher.	$\exists x$ [Phil(x)].

In predicative logic, we need to pay attention to the quantifies about their domain, order and use with negation, as illustrated by the following example.

Example 3.10. Domain, Order, and Negation When Using Quantifiers
Consider the mathematical statement: for any natural number n, either n is an even number, or n+1 is an even number. This statement expressed in natural language can be more concisely and precisely expressed in predicative logic as

$$\forall n\ [\text{Even}(n) \vee \text{Even}(n+1)],$$

where we use predicate Even(n) to mean n is an even number.

However, the above predicative logic expression does not consider "for any natural number". This can be compensated by explicitly indicating the **domain** of variable n associated with the universal quantifier, and the expression becomes:

$$\forall n \in \mathbb{N}\ [\text{Even}(n) \vee \text{Even}(n+1)],$$

where $\mathbb{N}$ represents the set of natural numbers.

In predicative logic, the **order** of the quantifiers is important. Look at the following two statements. The first is true while the second is false.

(1) $\forall x \in \mathbb{N}, \exists y \in \mathbb{N}\ (y = x + 1)$	Every natural number has a successor.
(2) $\exists y \in \mathbb{N}, \forall x \in \mathbb{N}\ (y = x + 1)$	There is a natural number which is the successor of all natural numbers.

With negation, we need to differentiate "Not-All" and "All-Not" statements. More precisely, consider the following two statements.

$\neg(\forall x \in \mathbb{N}\ [\text{Even}(n)])$	Not all natural numbers are even.

$\forall x \in \mathbb{N}\ [\neg \mathrm{Even}(n)])$ All natural numbers are not even.

The first statement (Not-All) happens to be true, and the second statement (All-Not) happens to be false.

Negation with quantifiers satisfies the following **negation properties**:

$$\neg(\exists x\, \mathrm{P}(x)) = \forall x \neg \mathrm{P}(x), \ \ \neg(\forall x\, \mathrm{P}(x)) = \exists x \neg \mathrm{P}(x).$$

For instance, the true statement

$\neg(\forall x \in \mathbb{N}\ [\mathrm{Even}(n)])$ Not all natural numbers are even

is equivalent to

$\exists x \in \mathbb{N}\ [\neg \mathrm{Even}(n)]$ There exists a natural number that is not even.

The false statement

$\forall x \in \mathbb{N}\ [\neg \mathrm{Even}(n)]$ All natural numbers are not even

is equivalent to

$\neg(\exists x \in \mathbb{N}\ [\mathrm{Even}(n)])$ There exists no natural number that is even.

The negation properties can be used in cascade. For instance, the statement

$\neg(\exists y \in \mathbb{N}, \forall x \in \mathbb{N}\ (y \neq x + 1))$ There is no natural number which is
not the successor of any natural number

is equivalent to

$\forall y \in \mathbb{N}, \exists x$
$\in \mathbb{N}\ (y = x + 1)$ Any natural number is the successor of some natural number.

This statement is false, as zero is not the successor of any natural number.

3.1.2.2 More Examples of Writing Predicative Logic Expressions

We discuss more examples to show how to write predicate logic expressions. In particular, we show how natural language statements can be expressed as predicate logic expressions. The latter can express the statements more rigorously.

Example 3.11. Representing Infinity
Consider the following statement expressed in natural language:

There exist infinitely many prime numbers.

How to represent this statement as a predicate logic expression?

We use the predicate Prime(m) to represent "m is a prime number". Now the key is how to express "infinite". There are several ways to do it.

The first way is to express "infinite" directly by interpreting it. We convert the original statement "there exist infinitely many prime numbers" into a more concrete statement: "for any natural number, these exists some prime number larger than it". Since there are infinitely many natural numbers, there are infinitely many prime numbers.

Then, "there exist infinitely many prime numbers" can be expressed by the following expression:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, [(m > n) \wedge (\text{Prime}(m))]$$

which is a direct rewriting of the more concrete statement.

The second way is to is to express "infinite" as "not finite". We first write a statement for "finite", and then negate it. The following two expressions are examples of this method. We use the idea: for any finite subset of N, there exists a maximum number in this subset.

$$\neg(\exists n \in \mathbb{N}, \forall m \in \mathbb{N}, [(m > n) \rightarrow \neg(\text{Prime}(m))])$$
$$\neg(\exists n \in \mathbb{N}, \forall m \in \mathbb{N}, [(\text{Prime}(m)) \rightarrow (m \leq n)])$$

We leave it as an exercise to show that the above two expressions indeed express the statement "there exist infinitely many prime numbers".

Example 3.12. Predicate Refinement
We use the predicate Prime(m) to represent "m is a prime number" in the above example, to obtain the following expression for "there exist infinitely many prime numbers":

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, [(m > n) \wedge (\text{Prime}(m))]$$

Let us try to express "prime number", by refining the predicate Prime(m). From mathematics, prime numbers are positive integers which have only two factors: 1 and themselves. This definition of prime number is a little bit difficult to express, because it contains the phrase "have only". We can use an equivalent but easier-to-express definition: a prime number is a natural number that cannot be generated by multiplying other two natural numbers greater than 1. Thus, we have

$$\mathrm{Prime}(m) = \forall \mathrm{p,q} \in \mathbb{N}, \mathrm{p,q} > 1(m \neq pq).$$

Here, p, q ∈ N, p, q > 1 is the domain of variable *p*, *q*. Intuitively, the above equation says that *m* is a prime number if *m* is not the product of two natural numbers *p* and *q* which are both greater than 1.

We can directly substitute Prime(*m*) in the original expression:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, [(m > n) \wedge (\forall p, q \in \mathbb{N}, p, q > 1(m \neq pq))].$$

But usually, we write all quantifiers in front of the expression. Thus, we have:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1[(m > n) \wedge (m \neq pq)].$$

Here, we emphasize again the importance of the order of the quantifiers. Consider the following two expressions

$$\forall n \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1, \exists m \in \mathbb{N}\ [(m > n) \wedge (m \neq pq)]$$

and

$$\exists m \in \mathbb{N}, \forall n \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1[(m > n) \wedge (m \neq pq)].$$

Neither expression is equivalent to the statement "there exist infinitely many prime numbers".

Now consider another related logic statement called the twin prime conjecture:

There are an infinite number of twin prime pairs.

How can we represent this statement as a predicate logic expression? The key here is what is "twin prime pair". Twin primes (or a twin prime pair) are two prime numbers with a difference of 2. For instance, 5 and 7 form a twin prime pair, so do 11 and 13. Adding this definition to the original expression for infinite prime numbers, we have a predicate logic expression for the twin prime conjecture as follows:

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, \forall p, q \in \mathbb{N}, p, q > 1[(m > n) \wedge (m \neq pq) \wedge (m + 2 \neq pq)].$$

Example 3.13. Representing Potentially Unbounded Process

Let us consider the following statement called the Collatz conjecture, which is a logic statement involving a potentially unbounded process.

For any positive integer *n*, multiply *n* by 3 and add 1 if *n* is odd, and divide *n* by 2 if *n* is even. Repeat this process and you will always get 1.

For example, for $n = 15$, the above process become $15 \rightarrow 46 \rightarrow 23 \rightarrow 70 \rightarrow 35 \rightarrow 106 \rightarrow 53 \rightarrow 170 \rightarrow 85 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. After 17 steps, the process converges to 1.

We use $f(n)$ to represent one step for integer n, and use $f^{(m)}(n)$ to represent the m times composition of function f, that is, $f\,(f\,(\cdots f\,(n)\cdots))$. The Collatz conjecture can be expressed by the following predicate logic expression:

$$\forall n, \exists m, \left[\, f^{(m)}(n) = 1 \right], \text{where} f(n) = \begin{cases} 3n + 1, \text{if } n \equiv 1 \pmod 2; \\ n/2, \quad \text{if } n \equiv 0 \pmod 2. \end{cases}$$

At present, this conjecture has not yet been solved.

3.1.2.3 Inference Rules and Axiomatic Systems in Boolean Logic

We have implicitly used **axioms** and **inference rules** from school logic in understanding the material of Boolean logic. On the other hand, we also point out that Boolean algebra is different from school algebra. This seemingly contradiction raises a question: is logic thinking different in computer science from that in ordinary mathematics? How to rigorously specify the difference?

Normally, logic thinking in computer science is the same as that in ordinary mathematics. More specifically, we can use a mathematic method called **axiomatic systems** to specify any particular logic system. An axiomatic system is built from three components: (1) a set of *elements and operators* on these elements, (2) a set of *axioms*, i.e., given properties about the elements and operators; and (3) a set of *inference rules* to derive new properties from known properties.

To rigorously specify a logic system in computer science, such as Boolean logic, that is different from ordinary school mathematics, we explicitly specify different operators and axioms but normally **use the same inference rules of mathematics**. For instance, comparing to algebra in high school mathematics, Boolean logic introduces a new NOT operator. In addition, as shown in Table 3.2, Boolean logic introduces three new axioms (the idempotence, the absorption, and the complementation laws) and changes the distributivity and the annihilator laws.

We explicitly list three sets of commonly used inference rules in Box 3.2. They are all inference rules of ordinary mathematics, and can be used to infer a statement (conclusion), given one or more statements (premises).

Box 3.2. Several Commonly Used Inference Rules

Modus Ponens:

Given	$X \to Y$	Every Web page has a URL
	$\underline{X \quad}$	My homepage is a Web page
Conclude	Y	My homepage has a URL

Modus Tollens:

Given	$X \to Y$	Every Web page has a URL
	$\underline{\neg Y \quad}$	My cellphone does not have a URL
Conclude	$\neg X$	My cellphone is not a Web page

Negating Quantified Predicate:

Given	$\neg(\exists x\, P(x))$	Given	$\forall x \neg P(x)$
Conclude	$\forall x \neg P(x)$	Conclude	$\neg (\exists x\, P(x))$
Given	$\neg(\forall x\, P(x))$	Given	$\exists x \neg P(x)$
Conclude	$\exists x \neg P(x)$	Conclude	$\neg (\forall x\, P(x))$

3.2 Automata and Turing Machines

When a computational process has a single step, Boolean logic often suffices to produce correct answer and ensures logic correctness. However, when a computational process involves multiple steps, we often prefer new models. A key concept is *automata*, also known as state machines. An automaton can remember things by holding states and use state transitions to represent steps.

David Hilbert (1862–1943) put forward a very fundamental and general problem that requires multi-step computational processes, the *Entscheidungsproblem* (the decision problem), i.e., mechanically proving theorems of mathematics. Alan Turing gave a negative answer to the decision problem, but in the process, proposed Turing machines, a class of automata that turn out to be able to solve any computable problems. Turing machines are key milestones and cornerstones of correctness and generality of computational processes.

Fundamental limitations of computation are also discussed. There exist incomputable problems that cannot be solved by any Turing machine. We also have Gödel's incompleteness theorem: being true and being provable are not the same thing. In any reasonably sophisticated mathematic system, there are mathematic theorems which cannot be proven.

3.2.1 Mechanical Theorem Proving

Mechanical theorem proving (also known as automated theorem proving or computer-assisted proof), requires that in the process of calculation or proof, after each step, there is a certain rule to choose the next step. Along this path, the process will finally reach the required conclusion. In this way, people hope to avoid those highly skilled mathematical calculations or proofs and replace them with the powerful computing power of modern computers.

The idea of mechanical proof can be traced back to the seventeenth century French mathematician Rene Descartes (1596–1650). Descartes once had a great idea: "All problems can be turned into mathematical problems; all mathematical problems can be turned into algebraic problems; and all algebraic problems can be turned into solving algebraic equations." Descartes created analytical geometry, established the bridge between the spatial form and the quantitative relationship, and established the framework to solve elementary geometric problems based on algebraic methods.

In 1928, David Hilbert stated the problem of mechanical theorem proving more clearly: given an axiomatic system, is there a mechanical method (now called an algorithm) that can verify the truth or falsity for every proposition in this system? In Sect. 3.2.3 we will see that the answer to this Entscheidungsproblem is No.

However, although it is impossible to use an algorithm to determine all the propositions, it is still feasible to use mechanized methods for specific problems in specific fields. For example, the elimination method (Wu's method) based on the zero-point set of the polynomial system proposed by Professor Wenjun Wu (also known as Wu Wen-tsün,) can be applied to the mechanical proof of a large number of geometric theorems.

The first major theorem proved with the help of computer is the four-color theorem. This famous four-color theorem in graph theory asserts that any planar graph can be 4-colored, that is, there is a way to dye each vertex with one of four colors so that any adjacent vertices do not have the same color. The four-color theorem was first proposed by Francis Guthrie (1831–1899) in 1852. This problem puzzled mathematicians for more than a 100 years. It was finally proved by Kenneth Appel and Wolfgang Haken in 1976 with the help of computer.

The idea of their proof is as follows: if a certain structure appears in the planar graph, this part can be replaced with a smaller structure (that is, reduce the size of the original graph), while the 4-colored property is unchanged. That is, if the new graph can be 4-colored, the original larger graph can also be 4-colored. For example, a vertex with a degree no greater than 3 can be removed, because it does not affect whether the whole graph can be colored by 4 colors. Appel and Haken proved that there are 1936 planar graphs that cannot be reduced to one another. Any other planar graph can always reach one of these 1936 graphs through the specific process of reduction. Finally, with the help of the computer, after more than 1000 h of calculation, they verified that all 1936 graphs can be 4-colored and thus proved the four-color theorem.

3.2.2 Automata

When a proof process is viewed as a computational process, it is usually a multistep process. We start from the axioms or a known true statement (a theorem), and repetitively apply the inference rules to arrive at new true statements and the final conclusion. The result of an inference step should be memorized as a state, and used as part of inputs for future steps. That is, we need a machine that holds states. Such a machine is called an **automaton**. We will introduce two classes of automata, namely finite state automata and Turing machines.

Consider a simple vending machine, which sells bottled water and bagged biscuits. The price of bottled water is \$1 per bottle and the price of biscuits is \$2 per bag. The vending machine accepts only \$1 or \$2 banknotes. In any state, a buyer can perform one of five actions to the vending machine: (1) insert a \$1 banknote, (2) insert a \$2 banknote, (3) press the "Buy water" button, and (4) press the "Buy biscuits" button, and (5) press the "Get money back" button.

Initially, the vending machine is in the state q_0 (initial state). If the buyer inserts a \$1 banknote, the vending machine will transfer to a new state q_1. If the buyer chooses to buy bottled water in state q_1, the vending machine will output one bottle of water and go back to state q_0. If the buyer inserts one more \$1 banknote in state q_1, the vending machine will transfer to a new state q_2. If the buyer chooses to buy something in state q_2, the vending machine will output the corresponding goods and go back to q_0 (biscuits) or q_1 (bottled water). If the buyer wants to get the money back in state q_1 or q_2, the vending machine will return the corresponding amount of money and go back to state q_0.

Figure 3.3 is called the **state-transition diagram**, which shows the above state transition rules of the vending machine. Note that the arrow notation specifies an input-output pair. "\$1→\$1" at the arrowed curve in state q_2 denotes "when the buyer inserts a \$1 bill, the machine outputs a \$1 bill and stays in state q_2." The notation "\$1→" at the arrowed curve from state q_0 to state q_1 denotes "when the buyer inserts a \$1 bill, the machine outputs nothing and transitions from state q_0 to state q_1."

The state-transition diagram can be equivalently written as a **state-transition table** in Table 3.10. Note that the state transition diagram happens to omit some possible transitions, while the state transition table lists all possible transitions.

The above computational model for the vending machine is called a (deterministic) **finite automaton**, also known as a finite-state automaton. It is a model of computation suitable for computational processes where only finite numbers of states are involved. A computational process that involves potentially infinite number of states cannot be modeled by a finite-state automaton. Finite automata cannot handle infinite states, as shown by the following example.

Example 3.14. Palindromes Cannot Be Recognized by Finite Automata
A palindrome is a character string that is the same when reading backwards. For instance, 1991 is a palindrome, so is 0100110001110000111100001110001100 10. We leave it as an exercise for students to show that palindromes cannot be recognized by finite automata.

☵

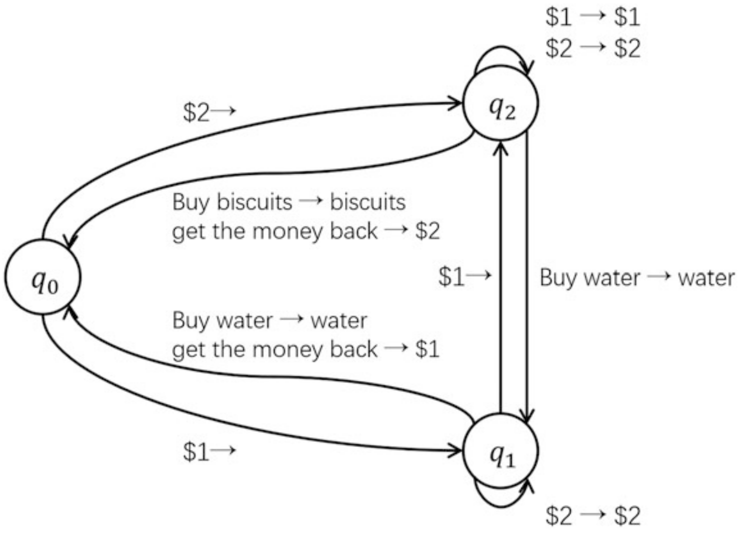


Fig. 3.3 The state transition diagram for a vending machine

Table 3.10 State transition table of a vending machine

Current state	Input	Output	Next state
q_0	Insert \$1	Null	q_1
	Insert \$2	Null	q_2
	Buy water	Null	q_0
	Buy biscuits	Null	q_0
	Get money back	Null	q_0
q_1	Insert \$1	Null	q_2
	Insert \$2	Output \$2	q_1
	Buy water	Output water	q_0
	Buy biscuits	Null	q_1
	Get money back	Output \$1	q_0
q_2	Insert \$1	Output \$1	q_2
	Insert \$2	Output \$2	q_2
	Buy water	Output water	q_1
	Buy biscuits	Output biscuits	q_0
	Get money back	Output \$2	q_0

3.2.3 Computation on Turing Machine

What is computable? Alan Turing gave a rigorous definition in his famous paper in 1936. His idea is that all the infinite mathematical entities, such as numbers, variables, functions and predicates, can be mapped to the infinite set of real numbers. A mathematical entity is computable if its corresponding real number is computable.

What is computable is reduced to what real numbers are computable. "The computable numbers [are] the real numbers whose expressions as a decimal are calculable by finite means." Another equivalent definition is: "A number is computable if its decimal can be written down by a machine."

Example 3.15. The Circular Constant π Is Computable
We use π to denote the circular constant (the ratio of circumference to diameter of any circle). It is an irrational number and has infinitely many decimal digits. Nevertheless, π is computable according to Turing's definition: π is a real number whose decimal digits are calculable by finite machines. That is, any sequence of digits of π that we want can be produced by finite means.

Suppose we want the first 800 digits of π. This sequence can be produced by the following finite means: running the following pi.go program[1] on a laptop computer. The program is finite, as it contains 27 lines of code. The computer is finite with 2GB memory capacity. The running time is finite, as executing the pi.go program consumes less than 1s. The entire execution process is automatic.

```
package main
import "fmt"
func main() {
  var r [2801] int
  var i, k, b, d int
  c := 0
  for i = 0; i < 2800; i++ {
    r[i] = 2000
  }
  for k = 2800; k > 0; k -= 14 {
    d = 0
    i = k
    for ;; {
      d += r[i] * 10000
      b = 2 * i - 1
      r[i] = d % b
      d /= b
      i--
      if i == 0 {break}
      d *= i
    }
    fmt.Printf("%.4d", c + d / 10000)
    c = d % 10000
  }
}
```

[1]This pi.go program is rewritten into Go code from a 160-character C program written by Dik T. Winter of the Centrum Wiskunde & Informatica (CWI) in the Netherlands. Dr. Ben Lynn of Stanford University analyzed the C code. Please see his analysis note at https://crypto.stanford.edu/pbc/notes/pi/code.html.

The program pi.go produces the first 800 decimal digits of π:

```
    3141592653589793238462643383279502884197169399375105820974944
4592305218781640628620899862803482534211706798214808651328230
6647093844609556423058223172535940812848111745028410270193852
1105559644622948954930388415196442881097566593344612847564823
3786783165271201909145648566923468758034861045432664821339360
7260249141273724587006606315588174881520923797096282925409171
5364367892590360011330530548820466521384146951941517064160943
3057270365759591953092186117381932611793105118548074462379962
4232749567351885752724891227938183011949129833673362440656643
0860213903734946395224737190702179860943702770539217176293176
7523846748184676609194051320005681271452635608277857713427577
8960917363717872146844090612249534301465495853710507922796892
5892354201995611212902196086403441815981362977477130996051870
7211349999998372978049951059731732816096318
```

In his 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem", Alan Turing described an abstract computer which was later called a Turing machine. Turing machines are a more powerful model of computation than finite automata.

The organization of a Turing machine is shown in Fig. 3.4. At a minimum, a Turing machine is comprised of three components: (1) an infinite tape, (2) a read/write head, and (3) a finite state-transition diagram in a finite state controller.

The tape has infinitely many squares (cells) extending to both directions. Each square contains a symbol, such as 0, 1 and blank. The blank symbol can be written explicitly as B to avoid confusion.

Initially, the tape contains the input string between two blanks. All other squares are blank. The head points to the first input symbol (or to the blank square left of the first input symbol). The state-transition table resides in the finite state controller.

The state-transition diagram or the equivalent state transition table governs the behavior of a Turing machine, as illustrated in Fig. 3.4 and Table 3.11. This particular Turing machine does a cleanup. It scans the input string from left to right, erases each 0 or 1 (replacing 0 or 1 by blank B), and stops when reads a @.

The machine starts at initial state q_0 and stops at final state q_1. Sometimes we explicitly name the final state q_1 as Halt, when there is just one final state. At each step, the head reads the symbol in the pointed square, writes an appropriate symbol, and moves the head to left or right, and then the machine transition to the next state.

Definition: A **Turing machine** is a 7-tuple $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{\text{Accept}}, q_{\text{Reject}}\}$.

- Q is a finite, non-empty set of states.
- Σ is a finite, non-empty set of input symbols.
- Γ is a finite, non-empty set of tape symbols. There is a special character $B \in \Gamma$ for the blank symbol. We require $B \notin \Sigma$ and $\Sigma \subset \Gamma$.
- $\delta : (Q - \{q_{\text{Accept}}, q_{\text{Reject}}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\rightarrow, \leftarrow\}$ is the transition function.
- $q_0 \in Q$ is the initial state.

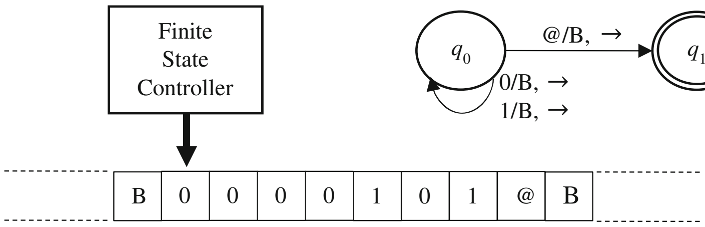


Fig. 3.4 Organization of a Turing machine, with a cleanup function example

Table 3.11 State transition table of a Turing machine

Current state	Symbol read	Symbol to write	Head move	Next state
q_0	0	B	→	q_0
q_0	1	B	→	q_0
q_0	@	B	→	q_1 (Halt)

- $q_{\text{Accept}} \in Q$ is the accept state.
- $q_{\text{Reject}} \in Q$ is the reject state.

To make the definition concrete, let us review again the cleanup Turing machine illustrated in Fig. 3.4 and Table 3.11. For this machine, $Q = \{q_0, q_1\}$, $\Sigma = \{0, 1, @\}$, $\Gamma = \{0, 1, @, B\}$. The initial state is q_0. The accept state is q_1 which means the machine successfully finishes the cleanup process. There is no reject state. The transition function δ is illustrated in Fig. 3.4 and Table 3.11.

If we look at the transition function more carefully, we may find that there is no definition of $\delta(q_0, B)$. When the computational task to erase the input string ends with the symbol @, it is impossible to read B in state q_0. However, it is always useful to write down the rule for all cases in order to handle exceptional conditions. One possible way to handle it is to set $\delta(q_0, B) = (q_2, B, \rightarrow)$ where q_2 is the reject state. This means if the end of the input is not @, the Turing machine will go to the reject state q_2 and stop.

Example 3.16. Palindromes Can Be Recognized by a Turing Machine
In Example 3.14, we claim that Palindromes cannot be recognized by finite automata. Here, we show it can be recognized by a Turing machine. Thus, Turing machines are more powerful than finite automata.

The Turing machine starts at state q_0 with the input string on the tape, enclosed between two blank squares. The machine has two final states. When the input string is not a palindrome, the machine eventually stops at state q_{Reject}, and outputs a 0 on the tape. When the input string is a palindrome, the machine eventually stops at state q_{Accept}, and outputs a 1 on the tape.

The input alphabet contains only two symbols: 0 and 1. The tape alphabet contains an additional symbol: the blank symbol B. The state transition table is shown in Table 3.12. Note that there are nine states, but only seven states trigger

Table 3.12 State transition table for a Turing machine to recognize any palindrome

Transition	Current state	Symbol read	Symbol to write	Head move	Next state
1	q_0	0	B	$\rightarrow$	q_{Seen0}
2	q_0	1	B	$\rightarrow$	q_{Seen1}
3	q_0	B	1	$\leftarrow$	$\boldsymbol{q}_{\textbf{Accept}}$
4	q_{Seen0}	0	0	$\rightarrow$	q_{Seen0}
5	q_{Seen0}	1	1	$\rightarrow$	q_{Seen0}
6	q_{Seen0}	B	B	$\leftarrow$	q_{Want0}
7	q_{Seen1}	0	0	$\rightarrow$	q_{Seen1}
8	q_{Seen1}	1	1	$\rightarrow$	q_{Seen1}
9	q_{Seen1}	B	B	$\leftarrow$	q_{Want1}
10	q_{Want0}	0	B	$\leftarrow$	q_{Back}
11	q_{Want0}	1	B	$\leftarrow$	$q_{\text{BackErase}}$
12	q_{Want0}	B	1	$\leftarrow$	$\boldsymbol{q}_{\textbf{Accept}}$
13	q_{Want1}	0	B	$\leftarrow$	$q_{\text{BackErase}}$
14	q_{Want1}	1	B	$\leftarrow$	q_{Back}
15	q_{Want1}	B	1	$\leftarrow$	$\boldsymbol{q}_{\textbf{Accept}}$
16	q_{Back}	0	0	$\leftarrow$	q_{Back}
17	q_{Back}	1	1	$\leftarrow$	q_{Back}
18	q_{Back}	B	B	$\rightarrow$	q_0
19	$q_{\text{BackErase}}$	0	B	$\leftarrow$	$q_{\text{BackErase}}$
20	$q_{\text{BackErase}}$	1	B	$\leftarrow$	$q_{\text{BackErase}}$
21	$q_{\text{BackErase}}$	B	0	$\leftarrow$	$\boldsymbol{q}_{\textbf{Reject}}$

state transitions. The machine stops when it enters any of the two final states. A final state does not trigger a state transition. In each of the seven states, the head may read one of three tape symbols 0, 1, or B. There are $3\times7=21$ transitions in Table 3.12.

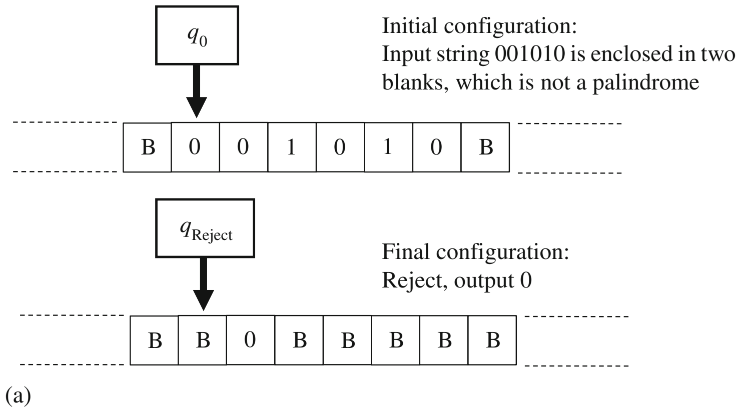
Figure 3.5 shows initial and final configurations of a Turing machine for recognizing palindromes, i.e., decides whether a string is a palindrome.

How does this Turing machine work? The basic idea is as follows.

- Iterate over the given the input string.
- When the first symbol and the last symbol match (they are both 0 or both 1), erase them and go to the next iteration.
- When the first symbol and the last symbol do not match, erase the remaining string and enter q_{Reject}.
- Until all symbols of the input string are all erased, then enter q_{Accept}.

For instance, consider the input string 001010, which is not a palindrome. The tape configurations of the iterations are shown below:

- Iteration 1:
  ...B**0**0101**0**B...; first and last symbols match, erase the two symbols and go to next iteration
- Iteration 2:
  ...BB**0**10**1**BB...; first and last symbols do not match, reject.



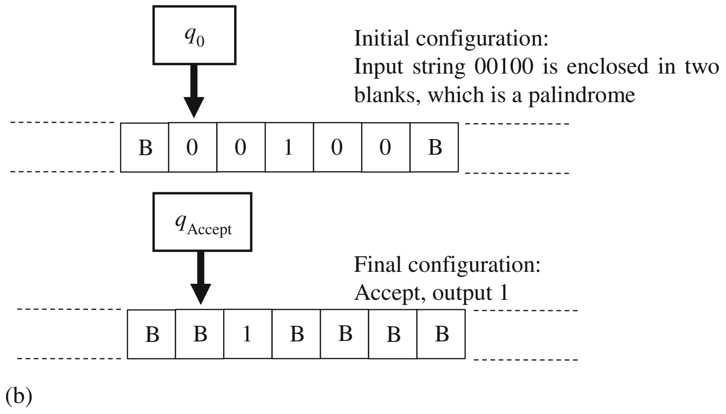


Fig. 3.5 Initial and final configurations of a Turing machine for palindrome recognition. (**a**) When the input string is 001010. (**b**) When the input string is 00100

Now, consider the input string 00100, which is a palindrome. The tape configurations of the iterations are shown below:

- Iteration 1:
  . . .B**0**010**0**B. . .; first and last match, erase and go to next iteration
- Iteration 2:
  . . .BB**0**1**0**BB. . .; first and last match, erase and go to next iteration
- Iteration 3:
  . . .BBB**1**BBB. . .; one symbol matches itself, erase it and go to next iteration
- Iteration 4:
  . . .BBBBBBB. . .; all symbols erased, output 1 and enter q_{Accept}.
- Final result . . .BBB1BBBB. . .

Applying the Turing machine definition to the palindrome-recognition problem, we have the following rigorous and concrete definition: a Turing machine recognizing palindromes is a 7-tuple $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{\text{Accept}}, q_{\text{Reject}}\}$, where

- $Q = \{q_0, q_{\text{Accept}}, q_{\text{Reject}}; q_{\text{Seen0}}, q_{\text{Seen1}}, q_{\text{Want0}}, q_{\text{Want1}}, q_{\text{Back}}, q_{\text{BackErase}}\}$, the three states before the semicolon are special states common to many Turing machines: $q_0 \in Q$ is the initial state, $q_{\text{Accept}} \in Q$ is the accept state, and $q_{\text{Reject}} \in Q$ is the reject state.
- The input alphabet is $\Sigma = \{0,\ \ 1\}$.
- The tape alphabet is $\Gamma = \{0,\ \ 1,\ \ \text{B}\}$.
- The transition function $\delta : (Q - \{q_{\text{Accept}}, q_{\text{Reject}}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{\rightarrow, \leftarrow\}$ is defined by Table 3.12.

The machine starts at q_0, with the head points to the leftmost symbol of the input string. When a 0 or 1 is read, the head writes a blank to the pointed square and moves to the right, and the machine transition to state q_{Seen0} or q_{Seen1} which means the machine has seen a 0 or a 1. In such a state, the machine moves the head to the right, until it reads a B, indicating that the head has just passed the rightmost symbol (the end of the string). The machine then transitions to state q_{Want0} or q_{Want1}, indicating the machine is expecting a 0 or 1 from the end of the string, to match the 0 or 1 seen. If the head reads a matching 0 or 1 in q_{Want0} or q_{Want1}, the machine erases it by writing a B and enters state q_{Back}, to go back to the beginning of the string and start the next iteration. If the head reads a mismatching symbol, e.g., reading a 1 in state q_{Want0}, the machine enters $q_{\text{BackErase}}$ to erase all remaining input symbols, and then enters q_{Reject} and halts. If the head reads a B in q_{Want0} or q_{Want1}, the machine has erased all matching 0's and 1's, thus the machine enters state q_{Accept} and halts

Note that in designing the state transition table, cares must be taken to ensure correct output, that is, a 1 is written on the tape when entering q_{Accept}, and a 0 is written on the tape when entering q_{Reject}.

Let us go through the step-by-step details of two small cases, for input strings 01 and 010, to verify the correctness of the Turing machine shown in Table 3.12.

For input string 01, which is not a palindrome, the sequence of transitions is shown in the following table, where each transition is a step in the computational process of deciding whether the input string is a palindrome. For each step, we list the tape contents before and after the transition, where the boldfaced symbol indicates the position of the read/write head. Before step 1, the tape contains B**0**1B and the head points to the square containing 0. The machine is at the initial state q_0, which triggers transition #1 in Table 3.12. After the transition, the symbol 0 is erased and the head moves right to point to the square containing 1.

Step	Before	Transition	After
1	B**0**1B	$<\#1, q_0, 0, B, \rightarrow, q_{\text{Seen0}}>$	BB**1**B
2	BB**1**B	$<\#5, q_{\text{Seen0}}, 1, 1, \rightarrow, q_{\text{Seen0}}>$	BB1**B**
3	BB1**B**	$<\#6, q_{\text{Seen0}}, B, B, \leftarrow, q_{\text{Want0}}>$	BB**1**B
4	BB**1**B	$<\#11, q_{\text{Want0}}, 1, B, \leftarrow, q_{\text{BackErase}}>$	B**B**BB
5	B**B**BB	$<\#21, q_{\text{BackErase}}, B, 0, \leftarrow, q_{\text{Reject}}>$	**B**0BB

For input string 101, which is a palindrome, the sequence of transitions is shown in the following table.

Step	Before	Transition	After
1	B**10**1B	$< \# 2, q_0, 1, B, \rightarrow, q_{\text{Seen1}}>$	BB**0**1B
2	BB**0**1B	$< \# 7, q_{\text{Seen1}}, 0, 0, \rightarrow, q_{\text{Seen1}}>$	BB0**1**B
3	BB0**1**B	$< \# 8, q_{\text{Seen1}}, 1, 1, \rightarrow, q_{\text{Seen1}}>$	BB01**B**
4	BB01**B**	$< \# 9, q_{\text{Seen1}}, B, B, \leftarrow, q_{\text{Want1}}>$	BB0**1**B
5	BB0**1**B	$< \# 14, q_{\text{Want1}}, 1, B, \leftarrow, q_{\text{Back}}>$	BB**0**BB
6	BB**0**BB	$< \# 16, q_{\text{Back}}, 0, 0, \leftarrow, q_{\text{Back}}>$	B**B**0BB
7	B**B**0BB	$< \# 18, q_{\text{Back}}, B, B, \rightarrow, q_0>$	BB**0**BB
8	BB**0**BB	$< \# 1, q_0, 0, B, \rightarrow, q_{\text{Seen0}}>$	BBB**B**B
9	BBB**B**B	$< \# 6, q_{\text{Seen0}}, B, B, \leftarrow, q_{\text{Want0}}>$	BB**B**BB
10	BB**B**BB	$< \# 12, q_{\text{Want0}}, B, 1, \leftarrow, q_{\text{Accept}}>$	B**B**1BB

3.2.3.1 Notable Details of Turing Machine

When learning Turing machines, students may experience several difficulties regarding details, which are summarized below.

Finite states. Any Turing machine has a finite number of states. Let us look at Table 3.12 again. The input string of palindrome can be of arbitrary length. However, the Turing machine has only 9 states. The same state transition table of 21 rows is used for input string of arbitrary length. It is a mistake to design a state transition table that depends on the length of the input string.

B $\boldsymbol{\notin \Sigma}$ **and B** $\boldsymbol{\in \Gamma}$. The input blank symbol B belongs to the tape alphabet but does not belong to the input alphabet. It is a mistake to confuse the blank symbol B with the capital letter B (0x42), the ASCII Space symbol (0x20), or the ASCII Null symbol (0x00). When the input string needs to contain such symbols, we can change the blank symbol notation to a new symbol such as β. Also note that the read/write head points to a *tape* square, which contains a symbol in the tape alphabet, including all input symbols *and* the blank symbol.

No stop in the middle. The Turing machine stops (halts) only when it enters a final state, either q_{Accept} or q_{Reject}. If it is at a non-final state, a transition will always be triggered and the machine will enter the next state, which could be the same state as the current one. However, the machine will never stop at a non-final state. The reason is that by the Turing machine definition, the transition function δ is a mathematical *function*, which means that δ is defined for every element of $(Q - \{q_{\text{Accept}}, q_{\text{Reject}}\}) \times \Gamma$. That is, for every non-final state s and tape symbol t, $\delta(s,t)$ is always defined, and there is always a next state to transition to.

To design a Turing machine for some specific computing problem, it is usually more intuitive for the novice to draw state-transition diagram, e.g., Fig. 3.4. However, one drawback of state-transition diagram is that it is easy to leave some $\delta(s,t)$

undefined. Though some transition seems impossible in the normal case, it is a good habit to write down the full transition function so as to handle the exceptional situations.

($|Q|$ − 2) × $|\Gamma|$ transitions. It follows from the above discussion that the state transition table of a Turing machine will always have ($|Q|$ − 2) × $|\Gamma|$ rows of transitions, where $|Q|$ is number of elements of set Q. The value 2 is for the two final states q_{Accept} and q_{Reject}. For example, the Turing machine in Table 3.12 has 9 states and its tape alphabet Γ has 3 elements. Thus, its state transition table has (9-2)×3=21 rows. Note that, the calculation only works if both accept state and reject state exist in the Turing machine.

Explicit and implicit input/output. For any Turing machine, the input string must explicitly appear in the tape between two blanks, before any step of state transition happens. The output of the computation is often defined as the string between the head-pointed square and the first blank right of it. Sometimes, we more carefully and explicitly define the output. For instance, in Example 3.16, we define the output to be a single-symbol string 1 for q_{Accept}, and string 0 for q_{Reject}.

One may also simplify the situation by doing the computation without cleanup, but assuming implicit output instead. In such a case, the output string may be mixed with a subset of symbols of input strings and intermediate results.

3.3 Power and Limitation of Computing

Real world problems can be either abstract (e.g., mathematic problems) or concrete (e.g., the problem of searching the Web). These problems can be formulated as computational problems for Turing machines.

Most of the problems one can imagine can be solved by Turing machines. For example, adding two integers, deciding whether an integer is a prime number, finding the most economic routes for a traveling salesman, etc. However, there exist problems that cannot be computed by any Turing machine. Some problems cannot even be effectively expressed for a Turing machine to solve. The computer science field has encountered paradoxes, incomputable problems, and incompleteness results. **Computability** is the subfield of computer science that studies the power and limitation of computers.

The existence of incomputable problems seems to be a negative fact. However, people have found ways to exploit such negative results for positive benefits. The following are some examples of ideas:

- Incomputable problems provide opportunities for human intelligence.
- Computationally hard problems can be used to design computer and Internet games.
- If a privacy protection technique can be formulated as incomputable problems, one cannot use computers alone to break privacy protection.

- In his recent book *Life after Google: The Fall of Big Data and the Rise of the Blockchain Economy*, the futurist and industry analyst George Gilder suggests that incomputability results by Kurt Gödel and Alan Turing provide a foundational piece for future technology systems.

3.3.1 Church-Turing Hypothesis

We have an important positive result called Church-Turing Thesis, due to Alonzo Church (1903–1995) and Alan Turing (1912–1954). Because it is actually a hypothesis, not a fully proven statement, it is also called Church-Turing Hypothesis. The thesis says that no reasonable abstract computer is more powerful than Turing machines. More specifically, we have the following results.

A problem is **Turing computable**, if there is a Turing machine that correctly solves the problem. That is, for any given input string, the Turing machine starting at the initial state q_0 will correctly stop at q_{Accept} or q_{Reject}.

We say a problem is a **computable problem**, if it is Turing computable. In other words, Turing machines are a general-purpose model for computability. If a problem is Turing incomputable, no other reasonable abstract computer can solve the problem, either. The generality statement that

```
Computable = Turing computable
```

can be viewed as a definition supported by many proven results.

Church-Turing Thesis: Assume a reasonable abstract computer X is given. Any problem computable in X is also Turing computable.

We say X is **reducible** to Turing machines. Church, Turing and other scholars have proven that many powerful models of computation are reducible to Turing machines. Their main method is to treat a problem as a mathematical function and simulate a step of abstract computer X by Turing machine steps.

A more recent result is the so-called **Polynomial Church-Turing Thesis**: If a problem is computable in abstract computer X and costs T_x steps, it is computable in a Turing machine and costs T_t steps, such that $T_t = \text{poly}(n, T_x)$. Here, n is the problem size. Intuitively, the thesis means that the Turing machine is at most polynomial times slower.

Consider the von Neumann model introduced in Chap. 2, which is a model of real computers such as a laptop computer. The von Neumann model can be augmented with infinite memory to obtain this result: **Turing machines are as powerful as a von Neumann computer** with infinite memory and arithmetic, logic, load, store, and conditional jump instructions. They can simulate each other with an overhead of no more than n^4.

Table 3.13 State transition table of a Turing machine

Current state	Symbol read	Symbol to write	Head move	Next state
q_0	0	B	$\rightarrow$	q_0
q_0	1	B	$\rightarrow$	q_0
q_0	@	B	$\rightarrow$	q_1 (Halt)
q_0	B	B	$\rightarrow$	q_0

3.3.2 (***) *Incomputable Problems and Paradoxes*

Computer science research also produced some seemingly negative results of computability. We discuss two such incomputable problems.

The halting problem. Given the description of an arbitrary Turing machine M and an input string x, decide whether M will terminate or run forever. "Turing machine terminates" means it eventually enters the accept state or the reject state.

The Entscheidungsproblem (the decision problem). Given a real number, decide if it is Turing computable. That is, if there is a Turing machine which can write down arbitrarily long decimal digits of the real number. If one wants n digits, for any n, the Turing machine will output the correct n digits and stop.

In this section, we show why the halting problem is not computable in Turing machine, and leave the Entscheidungsproblem as a thinking problem.

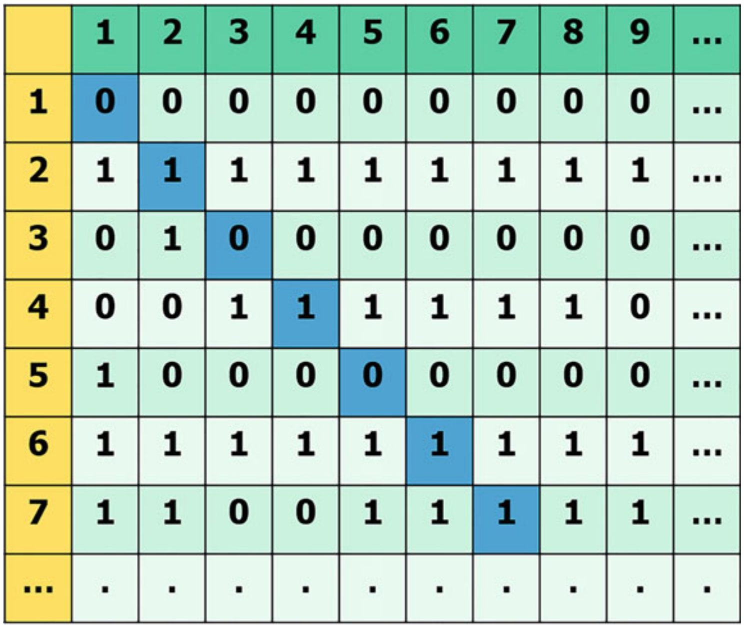
Example 3.17. The Halting Problem Is Not Turing Computable
We firstly give an example to show the case where a Turing machine may not terminate for some input string. Let us modify the machine in Fig. 3.4 and Table 3.11 a little bit. See Table 3.13 for the modified description of the state transition.

In this Turing machine, if the input string is 0000101@ as illustrated in Fig. 3.4, the machine will terminate in the halt state after reading the symbol @. However, if the input string is 0000101, the machine will never terminate and be stuck in state q_0. Thus, for the halting problem, if the input is this Turing machine and input string 0000101@, the answer should be "YES" or 1; while if the input string is 0000101, the answer should be "NO" or 0.

Before discussing the halting problem, let us first look at the representation of Turing machine more carefully. Any Turing machine can be represented by a 7-tuple $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{\text{Accept}}, q_{\text{Reject}}\}$, so it can be represented by a finite binary string. For example, we can write down the Turing machine in the normal way, like Q={q0, q1,q2}, and then translate it to ASCII code which is a finite binary string. Note, in such representation, not every finite binary string corresponds to a Turing machine, but it is easy to design a Turing machine which can decide whether a finite binary string corresponds to a Turing machine or not.

Thus, the set of all Turing machines is a subset of the set of all finite binary strings. This means the set of all Turing machines is countable. So is the set of all possible input strings.

We will prove the halting problem is not Turing computable by contradiction. Suppose there exists some Turing machine H which can compute halting problem.



	1	2	3	4	5	6	7	8	9	...
1	0	0	0	0	0	0	0	0	0	...
2	1	1	1	1	1	1	1	1	1	...
3	0	1	0	0	0	0	0	0	0	...
4	0	0	1	1	1	1	1	1	0	...
5	1	0	0	0	0	0	0	0	0	...
6	1	1	1	1	1	1	1	1	1	...
7	1	1	0	0	1	1	1	1	1	...
...	.	.	.	.	.	.	.	.	.	.

Fig. 3.6 Illustration of the halting problem

That is, for any Turing machine M, and for any input string x, $H(M, x) = 1$ or stops in the accept state if Turing machine M will terminate with the input string x; and $H(M, x) = 0$ or stops in the reject state if Turing machine M will never terminate (run forever) with the input string x. Thus H actually computes the following matrix in Fig. 3.6.

In this matrix, the i-th row represents the Turing machine whose binary representation is i, and the j-th column represents the input string j. If the Turing machine i can terminate with the input string j, the element of i-th row and j-th column in the matrix is 1; and if the Turing machine i cannot terminate with the input string j, the element of i-th row and j-th column in the matrix is 0. For some binary representation i, if there is no Turing machine corresponding to i, we set the i-th row to be all 1. Note that, in this matrix, we can list all possible Turing machines and all possible input strings. Of course, the matrix is infinitely large, with infinite number of rows and columns. But the number of rows and columns are both countable since the number of Turing machines and input strings are countable. Thus, the function of Turing machine H is actually to compute such a matrix.

Now, let us define a new Turing machine G based on H as follows (Fig. 3.7).

Since H exists, we can construct such Turing machine G. Now let us consider the case when the input string is G, the binary representation of the Turing machine G. For simplicity, we will use G to indicate both the Turing machine and its binary representation.

```
Turing machine $G$
Input string: binary string $i$
        Run $H(i,i)$;
        If $H(i,i) = 0$,
                Then, go to the accept state and halt
                Otherwise, run forever
```

Fig. 3.7 The definition of Turing machine G

Consider the element in the G-th row and G-th column in the matrix in Fig 3.6. If the element is 1, it means two things: (1) when the Turing machine G takes the input string G, it should halt after finite steps. This is due to the definition of the matrix. (2) $H(G, G) = 1$ due to the definition of H. However, examine the definition of G, we know when it takes input string G, it will run $H(G, G)$. Since $H(G, G) = 1$, the Turing machine G will run forever and never halt. Contradiction.

The other case is similar. If the element in the G-th row and G-th column in the matrix in Fig. 3.6 is 0, it also means two things: (1) when the Turing machine G takes the input string G, it should run forever. This is due to the definition of the matrix. (2) $H(G, G) = 0$ due to the definition of H. However, examine the definition of G, we know when it takes input string G, it will run $H(G, G)$. Since $H(G, G) = 0$, the Turing machine G will go to the accept state and halt. Contradiction again.

The element must take 0 or 1, and both cases will lead to contradiction. Thus, the assumption we made is not true. That is, there does not exist some Turing machine H which can compute the halting problem. Equivalently, the halting problem is not Turing computable.

One might think that the reason incomputable problems exist is because Turing machines are not powerful enough. There may exist other more powerful computational model which might be able to solve the halting problem or the decision problem. However, Church-Turing Hypothesis tells us this is not the case. From this viewpoint, Church-Turing Hypothesis is a negative result.

Some problems cannot even be effectively expressed as computational problems for Boolean logic or Turing machines to solve. A class of such problems are called paradoxes. We discuss two paradoxes below. The two cases involve self-reference and self-contradiction.

When encoding a real-world problem into the cyberspace, we need to be aware of paradoxes. We often need to change the problem specification to avoid any paradox. Sometimes, we can utilize paradox to create new functionality. An example of utilizing self-reference will be discussed in Sect. 5.3.2.

The liar paradox. When a liar says "This sentence is false", is he telling the truth? Remember the impatient guide problem in Examples 3.2 and 3.17. Suppose the guide comes from the Lying Village and makes the statement "What I am saying is false." Is the guide lying or not?

Such paradoxes contain a strange expression when expressed in Boolean logic. Let X stands for X=false. Sometimes we denote this naming as X: X=false. Now, is X true or false?

More generally, we may define Boolean variables using self-referencing expressions. An example is a pair of self-referencing expressions $\mathrm{Q} = \mathrm{Q} + \mathrm{S}$ and $\bar{Q} = \bar{Q} + \mathrm{R}$. In Sect. 5.3.2, we will see that these expressions with feedbacks are partial specification of the S-R latch, a new functionality to represent states. The contradiction is reconciled, when Q before and Q after the equality sign represent the next state and the current state, respectively.

The barber paradox. There is only one barber in a village who shaves all in the village who do not shave themselves. Does the barber shave himself?

A mathematic version of the barber paradox is **Russell's paradox** discovered by British scholar Bertrand Russell. In set theory, Russell's paradox considers the set *R* of all sets that are not members of themselves. Is *R* a member of itself? If we answer Yes (*R* is a member of itself), by the definition of *R*, *R* is NOT a member of itself. If we answer No (*R* is not a member of itself), by the definition of *R*, *R* must be a member of itself, since *R* is the set of **all** sets that are not members of themselves. Thus, whether we answer Yes or No, a contradiction will be the result.

Russell's paradox has contributed to the foundation of mathematics. In particular, it stimulated the creation of modern set theory (Zermelo–Fraenkel set theory), which is also a foundation of modern computer science.

3.3.3 (***) Gödel's Incompleteness Theorems

In 1928, David Hilbert proposed a suggestion for the solution to the foundational crisis of mathematics: to establish a set of axiom systems so that all mathematical propositions can be shown to be true or false in this system within a limited number of steps. The system Hilbert envisioned needs to answer the following questions:

- **Completeness:** for each of the true mathematical statements, we should be able to give a proof in this system.
- **Consistency:** there is no contradiction in the system, that is, there will be no statement that we can prove to be true and to be false in the same system.
- **Decidability:** we can find a way to determine whether a mathematical statement is true or false through only "mechanized" deduction.

For the general decidability question, i.e., the Entscheidungsproblem (the decision problem), Alan Turing's 1936 paper gave a negative answer.

How about the completeness and the consistency questions? In 1931, only 3 years after Hilbert's suggestion, Kurt Gödel (1906–1978) gave negative answers to these questions. The negative answers are called Gödel's incompleteness theorems.

Gödel's first incompleteness theorem: Any mathematical system that includes elementary number theory (natural numbers, addition, and multiplication) cannot have completeness and consistency at the same time.

Usually, researchers choose to sacrifice completeness in this dilemma. That is to say, for any reasonably sophisticated mathematical system, there exists some statement that is true, but we cannot prove it in this system. Some researchers have suggested that the "Goldbach's conjecture" may be the case. Here "reasonably sophisticated" means including the elementary number theory known as Peano Arithmetic shown in Box 3.3.

Box 3.3. Peano's Axioms of Arithmetic

In 1889, Giuseppe Peano (1858–1932) proposed Peano's axioms of arithmetic, later also called **Peano Arithmetic** for simplicity. This result has been widely used since then for mathematic logic in general and elementary number theory in particular. At a minimum, Peano Arithmetic consists of the following five axioms, and addition and multiplication operators can be defined based on these axioms.

1. Zero is a natural number; $0 \in \mathbb{N}$.
2. Every natural number has a successor in the set of natural numbers; $\forall n \in \mathbb{N}\ [\mathrm{S}(n) \in \mathbb{N}]$.
3. Zero is not the successor of any natural number; $\forall n \in \mathbb{N}\ [\mathrm{S}(n) \neq 0]$.
4. If the successors of two natural numbers are the same, then the two original numbers are the same; $\forall m, n \in \mathbb{N}\ [\mathrm{S}(m) = \mathrm{S}(n) \rightarrow m = n]$.
5. If a set contains zero and the successor of every natural number, then the set contains the set of natural numbers. This is called the induction axiom.

Gödel's second incompleteness theorem: For any mathematical system that includes elementary number theory, if it is consistent, then its consistency cannot be proved within itself.

In other words, whether there is a paradox in the system cannot be solved by relying on this system alone.

Gödel's incompleteness theorems deny Hilbert's proposal. Gödel's first incompleteness theorem tells us that truth and provability are two different things. A provable statement must be true if we stick to consistency, but a true statement is not necessarily provable since we sacrifice completeness. Gödel's second incompleteness theorem tells us that consistency cannot be proved within a mathematical system itself.

Example 3.18. A Case of Gödel's First Incompleteness Theorem

We discuss a specific statement to make our understanding of Gödel's first incompleteness theorem more concrete. The statement is Goodstein theorem, which is true, but cannot be proven in any mathematical system that includes elementary number theory.

We first need the concept of a Goodstein sequence.

Given any natural number n, its *hereditary base-b representation* is obtained as follows. First, write the sum of power base-b representation of n. If some exponent

m is greater than b, replace m by m's own sum of power base-b representation. Repeat this process until all numbers are less than or equal to b.

For instance, given n=266, its base-2 representation is $2^8 + 2^3 + 2$. There are two numbers >2, i.e., 8 and 3. Replacing them by their base-2 representations, we have $2^{2^3} + 2^{2+1} + 2$. Now we have only one number >2, which is 3. Replacing 3 with its base-2 representation, we have $2^{2^{2+1}} + 2^{2+1} + 2$. This is the hereditary base-2 representation of 266.

The change-of-base function $R_b(n)$ changes b to b+1 in n. That is, $R_b(n)$ replaces every b with b+1 in the hereditary base-b representation of n. For instance, for n=266, the hereditary base-2 representation is $2^{2^{2+1}} + 2^{2+1} + 2$. Changing every occurrence of 2 to 3, we have $R_2(266) = 3^{3^{3+1}} + 3^{3+1} + 3$, which is in the form of a hereditary base-3 representation of a much larger number:

$$R_2(266) = 443426488243037769948249630619149892887.$$

The *Goodstein sequence* for a given natural number n is denoted as $(n)_k$, where k ranges over the set of natural numbers, and the value of each $(n)_k$ is written as follows:

$$\begin{aligned}
&(n)_0 = n\\
&(n)_1 = R_2(n) - 1\\
&(n)_2 = R_3\big((n)_1\big) - 1\\
&\dots\\
&(n)_{k+1} = \begin{cases} R_{k+2}\big((n)_k\big) - 1 & \text{if } (n)_k > 0\\ 0 & \text{if } (n)_k = 0 \end{cases}.
\end{aligned}$$

For instance, for n=266, the Goodstein sequence is

$$\begin{aligned}
&(266)_0 = 2^{2^{2+1}} + 2^{2+1} + 2 && = 266\\
&(266)_1 = 3^{3^{3+1}} + 3^{3+1} + 2 && \approx 4.4 \times 10^{38}\\
&(266)_2 = 4^{4^{4+1}} + 4^{4+1} + 1 && \approx 3.2 \times 10^{616}\\
&\dots
\end{aligned}$$

A Goodstein sequence has three noteworthy properties:

- To go from one number to the next, two operations are performed. First, apply the change-of-base function, then subtract 1 from the result. The change-of-base function seems to significantly increase the number.
- The sequence grows tremendously fast. For instance, going from $(266)_1$ to $(266)_2$, the number grows to 3.2×10^{616}. Compare this to the fact that there are only about 10^{90} basic particles in the observable universe.

- The striking property is that this quickly growing sequence approaches 0! This is Goodstein theorem.

Goodstein theorem: Every Goodstein sequence always approaches 0.

That is, for any natural number n, there exists another natural number m, such that $(n)_m = 0$.

In 1944, Reuben Goodstein proved the Goodstein theorem. In 1982, Laurie Kirby and Jeff Paris showed that Goodstein theorem cannot be proven in any mathematical system that includes elementary number theory (i.e., Peano's Arithmetic).

Let us consider $(4)_k$ in more detail, i.e., the Goodstein sequence for number 4. Goodstein's function $G(n) = m$ is a function that maps n to m, where m is the smallest natural number such that $(n)_m = 0$. It is known that

$$G(4) = 3 \times 2^{402653211} - 3 \approx 6.895 \times 10^{121210694}.$$

It is impractical to compute all the non-zero items of the sequence. Our TA, Hongrui Guo, computed the first five, the largest five, and the last five items of the sequence before it reaches 0. These 15 values are as the following.

$$\begin{aligned}
&(4)_0 = 2^2 = 4\\
&(4)_1 = 3^3 - 1 = 2 \times 3^2 + 2 \times 3^1 + 2 = 26\\
&(4)_2 = 2 \times 4^2 + 2 \times 4^1 + 2 - 1 = 41\\
&(4)_3 = 2 \times 5^2 + 2 \times 5^1 + 1 - 1 = 60\\
&(4)_4 = 2 \times 6^2 + 2 \times 6^1 - 1 = 2 \times 6^2 + 6 + 5 = 83\\
&\ldots\\
&(4)_{max-2} \approx \text{1st1000DigitsOfMax}\\
&(4)_{max-1} \approx \text{1st1000DigitsOfMax}\\
&(4)_{max} \approx \text{1st1000DigitsOfMax}\\
&(4)_{max+1} \approx \text{1st1000DigitsOfMax}\\
&(4)_{max+2} \approx \text{1st1000DigitsOfMax}\\
&\ldots\\
&(4)_{G(4)-4} = 4\\
&(4)_{G(4)-3} = 3\\
&(4)_{G(4)-2} = 2\\
&(4)_{G(4)-1} = 1\\
&(4)_{G(4)} = 0
\end{aligned}$$

The sequence $(4)_k$ reaches the maximal value at index max. The maximal value is $(4)_{max} = 3.44754040154631\ldots.\ 10^{121210695}$. It is a natural number with 121210695

decimal digits. The first 1000 decimal digits of the largest numbers in the sequence are:
34475404015463100828681949798057549784788749379014868294825824
71181371747989862435944126537723138836357706343870670981471373770
12311972582711710423708488689955731916776345646600366175225653655
66707660736638221550278724966252575033308853486678486334114931903
14615269655969736866492160948225290436847365886709876147562092042
0058686649733118291758563321381202195171984182018123353393010562
71348711228774295067529019486998025111083607801145279169927216822
91424810789456193340854410358943085149505243047152149159691566503
1176899651610957212217360780656154707158846933785793375188967822
22962282279777804376115277338671809923516166212703892541980539479
29809819394864855229092228788578838875603483673163812706732806753
47382769219015375432616565108108081814310923711203313305968399971
67695778121779569475400362539158893903885373347987634477272363235
75017620929937195505529441455741049571737770925493092772866804132
388313424521454495162309274452555255771310652274759352993005306606
008562973170880130089684337752.

3.4 Exercises

1. What is NOT a possible truth value of proposition formula $P \vee Q$?

   (a) 0
   (b) 1
   (c) Either 0 or 1
   (d) Both 0 and 1

2. What is NOT a possible truth value of proposition formula $(P \vee \neg Q) \rightarrow P$?

   (a) 0
   (b) 1
   (c) Either 0 or 1
   (d) Both 0 and 1

3. Let the proposition formula G be $P \rightarrow Q$. How many assignments of the truth value to P, Q are there to make G **false**?

   (a) 1
   (b) 2
   (c) 3
   (d) 4

4. Let the proposition formula G be $(\neg Q \vee R) \leftrightarrow (\neg P \wedge R)$. How many assignments of the truth value to P, Q, R are there to make G **false**? Here, $A \leftrightarrow B$ is defined as $(A \rightarrow B) \wedge (B \rightarrow A)$

   (a) 2
   (b) 3
   (c) 4
   (d) 5

5. Write down the truth table of two proposition formulae $P \rightarrow$Q and $\neg P \vee Q$ and show that they are equivalent formulae.
6. Write down the disjunctive normal form of the two proposition formulae $P \vee Q$ and $P \wedge Q$.
7. Write down the disjunctive normal form of the proposition formula $P \vee \neg P$.
8. The conjunctive normal form of proposition formula $\neg(P \rightarrow Q)$ is

   (a) $(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg Q)$
   (b) $P \vee \neg Q$
   (c) $(P \vee Q) \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q)$
   (d) $P \wedge \neg Q$

9. In the theorem of disjunctive normal form, why do we need the assumption $F(x_1, x_2, \ldots, x_n) \not\equiv 0$?
10. Which of the following formula is not a tautology? Tautology refers to the proposition formula that is **true** in every possible assignment.

    (a) $(P \oplus P) \leftrightarrow (Q \wedge \neg Q)$
    (b) $(P \oplus \neg P) \leftrightarrow (Q \vee \neg Q)$
    (c) $((P \vee Q) \rightarrow P) \leftrightarrow (R \rightarrow R)$
    (d) $((P \wedge Q) \rightarrow P) \leftrightarrow (R \rightarrow R)$

11. Which of the following formula is a tautology?

    (a) $(P \rightarrow Q) \leftrightarrow (Q \rightarrow P)$
    (b) $(P \rightarrow Q) \leftrightarrow (\neg Q \rightarrow \neg P)$
    (c) $(P \rightarrow Q) \leftrightarrow (\neg Q \rightarrow P)$
    (d) $(P \rightarrow Q) \leftrightarrow (Q \rightarrow \neg P)$

12. Which of the following equation about "exclusive or" is correct?

    (a) $(x \oplus y) \wedge z = (x \wedge z) \oplus (y \wedge z)$
    (b) $(x \oplus y) \vee z = (x \vee z) \oplus (y \vee z)$
    (c) $\neg(x \oplus y) = (\neg x) \oplus (\neg y)$
    (d) $(x \vee y) \oplus z = (x \oplus z) \vee (y \oplus z)$

13. How many different Boolean functions of 4 variables are there?

    (a) 16
    (b) 32
    (c) 65,536
    (d) 4,294,967,296

14. Every playing card has two sides. One side is a number and the other side is a letter. Now there are four cards on the table, with A, 3, S, 8 facing up. **In the worst case**, how many cards do you need to turn over to confirm whether the following proposition is true for these four cards: the number on the vowel card (cards with letters AEIOU) must be even.

    (a) 3
    (b) 2
    (c) 1
    (d) 4

15. Three people, Alice, Bob and Charlie, said the following three sentences.

    - Alice: Both Bob and Charlie lie.
    - Bob: I tell the truth.
    - Charlie: Bob lies.

    Which of the following choices must be true?

    (a) Charlie lied.
    (b) Alice lied.
    (c) Bob lied.
    (d) All the previous three choices may be false.

16. Is the following logic correct? That is, assuming that the premise is true, is the conclusion also true? Please explain your answer.

    - Premise (1): students who take the course of Introduction to Computer Science can master Golang.
    - Premise (2): Some students who master Golang can serve as the teaching assistants in the course of Introduction to Computer Science next year.
    - Conclusion: some students who take the course of Introduction to Computer Science can serve as teaching assistants next year.

17. Denote by P the statement "I will travel around the world" and Q the statement "I have enough money". Let *f* be the statement "I will travel around the world, only if I have enough money". Which is the correct symbolization of *f*?

    (a)
    $$Q \rightarrow P$$
    (b)
    $$P \rightarrow Q$$
    (c)
    $$P \leftrightarrow Q$$
    (d)
    $$\neg P \vee \neg Q$$

18. Let $P(x)$ denote the statement "x masters Golang", $Q(x)$ denote the statement "x take the course of Introduction to Computer Science", and $R(x)$ denote the statement "x can serve as teaching assistants in the course of Introduction to Computer Science next year". Let f be the statement "students who take the course of Introduction to Computer Science can master Golang" and g be the statement "some people who master Golang can serve as teaching assistants in the course of Introduction to Computer Science next year". Which is the correct symbolization of f and g?

    (a)
    $$f : \forall x(Q(x) \wedge P(x)); g : \exists x(P(x) \wedge R(x))$$

    (b)
    $$f : \forall x(Q(x) \rightarrow P(x)); g : \exists x(P(x) \rightarrow R(x))$$

    (c)
    $$f : \forall x(Q(x) \rightarrow P(x)); g : \exists x(P(x) \wedge R(x))$$

    (d)
    $$f : \forall x(Q(x) \wedge P(x)); g : \exists x(P(x) \rightarrow R(x))$$

19. Can a Turing machine stop in the middle? Select the correct answer.
    (a) No. A Turing machine stops when it enters an accept state or a reject state.
    (b) Yes. A Turing machine can stop before it enters an accept state or a reject state, because the head sees a symbol not recognizable.
    (c) Yes. A Turing machine can stop before it enters an accept state or a reject state, because the machine enters a state with no next state.
    (d) Yes. A Turing machine can stop before it enters an accept state or a reject state, because the state transition table has only a finite number of entries, which cannot model all possible state transitions.

20. Design a Turing machine to accept the language $L = \{0^n \mid n \geq 1\}$ where the input alphabet is $\Sigma = \{0, 1\}$ and B represents the blank symbol. That is, the Turing machine should accept 0 or 000, but reject 010 or 100.
21. Design a Turing machine to accept the language $L = \{0^a1^b2^c \mid a, b, c \geq 0, a + b = c\}$ where the input alphabet is $\Sigma = \{0, 1, 2\}$. That is, the Turing machine should accept 0122 or 02, but reject 012 or 1002.
22. (***) In the definition of Turing machine, if the transition function is specified as $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\rightarrow\}$, which means that the Turing machine can only move its head to the right and cannot move its head to the left in each state, we call it a **right-moving Turing machine**. Which of the following propositions about right-moving Turing machine and Turing machine is correct?
    (a) There is a computing task which can be decided by Turing machine, but not by right-moving Turing machine.

(b) There is a computing task which can be decided by the right-moving Turing machine, but not by Turing machine.
(c) Right-moving Turing machine and Turing machine have the equivalent computing power.
(d) None of the above three propositions has been proved at present.

23. (***) Use the Pumping Lemma to prove that palindromes cannot be recognized by finite automata.

We say that an automaton recognizes the language of all palindromes, if when a character string of finite length is fed to the automaton as input, the automaton will finish in finite number of steps and output 1 if the string is a palindrome, and 0 if the string is not a palindrome.

Pumping Lemma. Let L be a language recognized by finite automata. Then there exists an integer n depending only on L such that every string $w \in L$ of length at least p (called the "pumping length") can be written as $w = xyz$ (w can be divided into three substrings), satisfying the following conditions:

(a) $|y| \geq 1$
(b) $|xy| \leq p$
(c) $\forall k \geq 0, xy^k z \in L$

24. Regarding the Church-Turing Hypothesis, which of the following is correct?

(a) The Hypothesis shows the generality feature of logic thinking. It says that Turing machine is a general-purpose model of computation.
(b) The Hypothesis says that Turing machine is not as general purpose as my laptop computer, because a Turing machine cannot create a PowerPoint presentation file.
(c) The Hypothesis says that Turing machine is general-purpose. Thus, one can use Turing machine to automatically prove the Goodstein theorem.
(d) The Hypothesis says that Turing machine and my laptop computer have equal power, in terms of computability.

3.5 Bibliographic Notes

The chapter quotation is from Professor Georg Gottlob of Oxford University, in a keynote speech addressing the 2009 European Computer Science Summit [1]. Kleene logic and the number of Kleene expressions are discussed in [2]. Kirby and Paris showed that Goodstein's theorem [3] cannot be proven in a mathematical system containing Peano Arithmetic [4].

References

1. Gottlob G (2009) Computer science as the continuation of logic by other means. Keynote Address, European Computer Science Summit
2. Mukaidono M (1982) New canonical forms and their application to enumerating fuzzy switching functions. In Proceedings of 12th international symposium on multiple-valued logic, pp 275–279
3. Goodstein RL (1944) On the restricted ordinal theorem. J Symbol Logic 9(2):33–41
4. Kirby L, Paris J (1982) Accessible independence results for Peano arithmetic. Bull Lond Math Soc 14:285–293

Chapter 4
Algorithmic Thinking

So if an algorithm is an idealized recipe, a program is the detailed set of instructions for a cooking robot preparing a month of meals for an army while under enemy attack.
—Brian Kernighan, 2017

Algorithmic thinking is concerned with solving problems **smartly**, by designing and using algorithms. We look at the world through an algorithmic lens.

A problem is specified by rigorously specifying the input and the desired output. An algorithm is a set of rules specifying the sequences of computational steps for solving a specific problem. That is, for any given input data, the algorithm produces the desired output data. Thus, an algorithm is specified as follows:

A Specific Algorithm
- **Input**: specifying the given input data.
- **Output**: specifying the desired output data.
- **Steps**: specifying the sequence of computational steps.

What exactly does *smart* mean in solving problems *smartly*? The following four characteristics of algorithmic thinking are noteworthy. Discussing these four characteristics constitutes the main contents of this chapter.

- A smart way to **define** algorithms. Donald Knuth gives a five-point definition of algorithms. Here, smartness manifests as simplicity. This definition captures the essence of algorithms, is extremely simple, yet universally applicable. Also, the simple definition makes it easy to check if a sequence of steps is an algorithm.
- A smart way to **measure** algorithms. We use asymptotic notations and asymptotic analysis methods to measure and analyze the time and space complexities of algorithms. This asymptotic way avoids many irrelevant details and idiosyncrasies. It also reveals an important division of the hardness of computational problems: the tractable (called P) and the intractable (called NP).
- Smart paradigms to **design** algorithms. We discuss several representative paradigms to reveal concrete skills and crafts, including divide-and-conquer, dynamic programing and greedy paradigms. They help design clever and much faster algorithms.

Z. Xu, J. Zhang, *Computational Thinking: A Perspective on Computer Science*,
https://doi.org/10.1007/978-981-16-3848-0_4

- Smart variations to **adapt** for problem nuances. Here, smartness manifests as flexibility. Problem nuances are utilized to increase algorithmic efficiency.

4.1 What Are Algorithms

We first discuss the algorithm definition and how to measure algorithms. The bubble sort algorithm is used as an illustrative example. In Sects. 4.2 and 4.3, we introduce the design and analysis of some representative algorithms.

4.1.1 Knuth's Characterization of Algorithm

In his seminal work *The Art of Computer Programming*, Donald Knuth proposed a five-point definition of algorithm, which has been widely accepted and used.

Definition. An **algorithm** is a finite set of rules specifying sequences of computational steps for solving a given problem, with the following five properties.

- *Finiteness*. An algorithm must always terminate after a finite number of steps.
- *Definiteness*. Each step of an algorithm must be precisely defined, that is, the actions to be carried out must be rigorously and unambiguously specified.
- *Input*. An algorithm has zero or more inputs, given before the algorithm begins or during the algorithm's execution.
- *Output*. An algorithm has one or more outputs, which relate to the inputs.
- *Effectiveness*. Every operation of an algorithm must be sufficiently rudimentary, such that in principle, the operation can be done by a human using paper and pencil, in finite time.

From the algorithmic lens, a problem is often specified as follows: design an algorithm according to Knuth's definition, such that for any given input data, it produces the desired output data. The algorithm is specified as follows, where the Steps part must satisfy the five properties in Knuth's definition.

- **Input**: specifying the given input data.
- **Output**: specifying the desired output data.
- **Steps**: specifying one or more sequences of computational steps.

Students can use a programming language to specify an algorithm. In fact, such a specification is more than a specification, but also an implementation of the algorithm, because the program can be compiled and automatically executed on a computer.

However, the chapter quotation tells us that an algorithm is not the same as a program. The quicksort algorithm was discovered before the invention of the Go programming language. Many algorithms were designed and used long before the modern computer era.

Algorithms represent essential ideas of programs. They are sufficiently detailed (Knuth's five points) to ensure that they are step-by-step procedures, but ignore many syntactic and semantic details of any particular programming language. In the design and analysis of algorithms, people often use **pseudocode**, i.e., some form of high-level natural language mixing mathematic notations, to specify an algorithm. This chapter follows this practice.

Example 4.1. Algorithm Versus Non-algorithm
Consider the problem of finding a common divisor of two positive integers x and y. The problem is easily specified:

- **Input**: Two positive integers x and y.
- **Output**: A positive integer z such that $x \% z = 0$ and $y \% z = 0$.

For instance, for input numbers x=36 and y=24, a desired output is 3. Indeed, the positive integer 3 is a common divisor, since 24%3=0 and 36%3=0.

One may devise many sequences of computational steps to solve this problem. However, a sequence of computational steps is not necessarily an algorithm. An algorithm must satisfy Knuth's five properties. Let us contrast two specifications.

The first specification (CD1) randomly picks a positive integer z and checks to see if it is a common devisor of x and y.

CD1: Randomly Pick and Check
- **Input**: Two positive integers x and y.
- **Output**: A positive integer z such that $x \% z = 0$ and $y \% z = 0$.
- **Steps**:

```
while true
  randomly pick a positive integer z
  if (x % z == 0) and (y % z == 0) then halt
```

However, CD1 is not an algorithm because it violates some of the five properties.

- It may never stop, violating the finiteness property.
- The step "randomly picking a positive integer" is not sufficiently rigorous or unambiguous. Out of the set of infinitely many positive integers, what is the meaning of "randomly picking"? It violates the definiteness property.

The second specification (CD2) is a revised version of **Euclid's algorithm**. The idea is to repetitively replace the larger of x and y by y and $x \% y$, till $y = 0$.

CD2: Euclid's Algorithm
- **Input**: Two positive integers x and y such $x > y$.
- **Output**: A positive integer z such that $x \% z = 0$ and $y \% z = 0$.
- **Steps**:

```
while $y \neq 0$
  $x, y = y, x \% y$
$z = x$
```

CD2 is indeed an algorithm. In fact, it does more than finding a common divisor, but finding the **greatest common divisor** of x and y, i.e., gcd(x, y). We leave it as exercises to show that CD2 indeed satisfies Knuth's five properties, and the algorithm finds gcd(x, y)=12, given two inputs x=36 and y=24.

4.1.2 The Sorting Problem and the Bubble Sort Algorithm

The sorting problem is a classic problem in computer science. The purpose of sorting is to adjust a sequence of "out-of-order" numbers into an ordered sequence of numbers. For simplicity, we assume that all positive integers are stored in an array, these integers have different values, and we need to sort these positive integers from small to large. More formally, the sorting problem is:

The Sorting Problem

- **Input**: a sequence $<a_1, a_2, \ldots, a_n>$ of n positive integers.
- **Output**: a reordered sequence $< a'_1, a'_2, \ldots, a'_n >$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

People have developed various algorithms to solve the sorting problem, such as bubble sort, insertion sort, quicksort, merge sort, heap sort, etc. They vary in simplicity, efficiency, and suitability to different application scenarios. They also provide rich examples for the design of algorithms. In this section, we discuss the bubble sort algorithm as an example to appreciate how an algorithm works. In Sect. 4.2, we will discuss insertion sort and merge sort to show the power of the divide-and-conquer strategy. In Sect. 4.3, we will further introduce the quicksort algorithm which is more sophisticated.

Example 4.2. The Bubble Sort Algorithm

The name "bubble sort" comes from the fact that large numbers will gradually bubble up to the top of the sequence through comparison and exchange operations, just like bubbles rising from the bottom in a water tank. The algorithm works as follows (Fig. 4.1).

The idea of bubble sort is very simple. In each round, compare every adjacent pair of numbers from left to right, and exchange the two numbers of a pair if the larger one is on the left side of the smaller one. After one round, the largest number will be moved to the rightmost position. We then go to the next round and compare-and-exchange every pair of numbers from left to right again.

For input A=[6, 2, 4, 1, 5, 9], the algorithm's sequence of execution steps is shown in Table 4.1. The output is A=[1, 2, 4, 5, 6, 9].

```
Input: An array A of length n to be sorted, e.g., A=[ 6, 2, 4, 1, 5, 9 ].
Output: A sorted array A, e.g., A=[1 , 2, 4, 5, 6, 9].
Steps:
  for i = 1 to n-1                          // for each round
        for j = 1 to n-i                    // compare every adjacent pair
            if A [j]> A [j + 1] then exchange A [j] with A [j + 1];
```

Fig. 4.1 The bubble sort algorithm

Table 4.1 Bubble sort [6, 2, 4, 1, 5, 9] into [1, 2, 4, 5, 6, 9]

Outer loop	Inner loop	State before	State after
First round	1st comparison 6>2, exchange	6, 2, 4, 1, 5, 9	2, 6, 4, 1, 5, 9
	2nd comparison 6>4, exchange	2, 6, 4, 1, 5, 9	2, 4, 6, 1, 5, 9
	3rd comparison 6>1, exchange	2, 4, 6, 1, 5, 9	2, 4, 1, 6, 5, 9
	4th comparison 6>5, exchange	2, 4, 1, 6, 5, 9	2, 4, 1, 5, 6, 9
	5th comparison 6<9, no exchange	2, 4, 1, 5, 6, 9	2, 4, 1, 5, 6, 9
Second round	1st comparison 2<4, no exchange	2, 4, 1, 5, 6, 9	2, 4, 1, 5, 6, 9
	2nd comparison 4>1, exchange	2, 4, 1, 5, 6, 9	2, 1, 4, 5, 6, 9
	3rd comparison 4<5, no exchange	2, 1, 4, 5, 6, 9	2, 1, 4, 5, 6, 9
	4th comparison 5<6, no exchange	2, 1, 4, 5, 6, 9	2, 1, 4, 5, 6, 9
Third round	1st comparison 2>1, exchange	2, 1, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	2nd comparison 2<4, no exchange	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	3rd comparison 4<5, no exchange	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
Fourth round	1st comparison 1<2, no exchange	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	2nd comparison 2<4, no exchange	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
Fifth round	1st comparison 1<2, no exchange	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9

Let us take another look at the bubble sort algorithm from the viewpoint of Knuth's characterization. It is indeed an algorithm according to Knuth's definition. The description of the algorithm defines a finite set of rules for specifying the sequence of computational steps to solve the sorting problem.

This specification satisfies the five properties in Knuth's definition of algorithms.

- *Finiteness*. In the bubble sort algorithm, the outer loop needs to be executed $n - 1$ times; for the i-th round, the inner loop contains $(n - i)$ comparisons and at most $(n - i)$ exchange. Therefore, the algorithm must terminate within $\sum_{i=1}^{n-1} \times (n - i) = \frac{n(n-1)}{2}$ steps.
- *Definiteness*. The meaning of each step in the bubble sort algorithm is very clear.
- *Input*. There are two inputs. One is the array A to be sorted, and the other is the length n of the array.
- *Output*. The output is the sorted array A, which share space with the input.
- *Effectiveness*. The basic operations of bubble sort are comparison and exchange. Both operations are sufficiently rudimentary. People can use pen and paper to achieve these operations accurately.

The bubble sort algorithm is inefficient, requiring roughly $n^2/2$ comparison operations. However, the algorithm has the obvious advantage of simplicity. It consists of a straightforward double loop and an easy-to-understand loop body. In addition, it has the *robustness* advantage: in the case when a small number of errors of comparison operations may occur, the resulting output will still be a mostly sorted sequence, since the algorithm does comparisons only on adjacent numbers.

4.1.3 Asymptotic Notations

It is always important to know whether an algorithm is efficient or not. Given a problem or an algorithm, how much resource (such as execution time or storage space) is theoretically required? For example, in the bubble sort algorithm in Example 4.2, for for n positive integers, the algorithm requires $n(n-1)/2$ comparisons and at most $n(n-1)/2$ exchange steps. Usually, we do not need to know the exact number or quantity of resource required. We can say the bubble sort algorithm requires roughly n^2 steps, or more professionally, we say the time complexity of bubble sort algorithm is $O(n^2)$. Here, $O(n^2)$ is the **asymptotic notation** of "exactly $n(n-1)/2$ comparisons and at most $n(n-1)/2$ exchange steps".

We usually use the asymptotic notations such as $O(\cdot)$, $o(\cdot)$, $\Omega(\cdot)$ to describe the efficiency of the algorithms or problems. The following is the formal definition of asymptotic notations.

Definition: let f, $g : \mathbb{N} \rightarrow \mathbb{N}$ be two functions, where $\mathbb{N}$ is the set of natural numbers.

Table 4.2 Equalities and inequalities regarding o, O, Ω, Θ notations

Notation	Equalities and Inequalities		
o	$n^{1.58} = \mathrm{o}(n^2)$	$n^{1.58} \neq \mathrm{o}(n^{1.58})$	$n^2 \neq \mathrm{o}(n^{1.58})$
O	$n^{1.58} = \mathrm{O}(n^2)$	$n^{1.58} = \mathrm{O}(n^{1.58})$	$n^2 \neq \mathrm{O}(n^{1.58})$
Ω	$n^{1.58} \neq \Omega(n^2)$	$n^{1.58} = \Omega(n^{1.58})$	$n^2 = \Omega(n^{1.58})$
Θ	$n^{1.58} \neq \Theta(n^2)$	$n^{1.58} = \Theta(n^{1.58})$	$n^2 \neq \Theta(n^{1.58})$

- $f(n) = O(g(n))$ if $\exists$ constant $c, d > 0, \ \forall n > d, f(n) \leq cg(n)$.
- $f(n) = o(g(n))$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) = \Omega(g(n))$ if $\exists$ constant c, d > 0, $\forall n > d, f(n) \geq cg(n)$.
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Intuitively, these notations have the following *asymptotic* meanings:

- The big-O notation denotes that g is an *upper bound* of f;
- The small-o notation denotes that g is a *strict upper bound* of f;
- The Ω notation denotes that g is a lower bound of f; and
- The Θ notation denotes that g is the same order of f.

It is a good learning practice to compare these notations in one place with some concrete values, to see their differences. For instance, given $f(n) = n^{1.58}$ and $g(n) = n^2$, we have the equalities and inequalities shown in Table 4.2.

The above table reveals something interesting regarding equality when expressing asymptotic values: the commutativity law and the transitivity law of ordinary math do not hold anymore. It is correct to write $n^{1.58} = \mathrm{O}(n^2)$, but incorrect to write $\mathrm{O}(n^2) = n^{1.58}$ or $n^2 = \mathrm{O}(n^{1.58})$. In fact, the following equalities hold. However, $\mathrm{O}(n^4) \neq \mathrm{O}(n^{1.6})$.

$$n^{1.58} = \mathrm{O}\left(n^4\right),$$

$$n^{1.58} = \mathrm{O}\left(n^3\right),$$

$$n^{1.58} = \mathrm{O}\left(n^2 + n - 3\right),$$

$$n^{1.58} = \mathrm{O}\left(n^{1.6}\right).$$

The meaning of the equal sign (=) has changed with asymptotic notations. It becomes *single-direction equality*. The equation $n^{1.58} = \mathrm{O}(n^2)$ means that the right side value n^2 is an upper bound of the left side value $n^{1.58}$..

The introduction of the asymptotic notations helps us focus on the dominant term when n becomes large. Though $n^2 = O(2n^2 + 3n - 4)$ is correct according to the definition of big-O notation, it is strange to use the notations in this way. Usually, we will say $2n^2 + 3n - 4 = O(n^2)$ where $O(n^2)$ represents the main term of function $2n^2 + 3n - 4$ and it helps us focus on how fast the function grows with the input size n. Let us analyze the bubble sort algorithm as an example.

Example 4.3. Time Complexity of the Bubble Sort Algorithm
To sort n positive integers, we know the bubble sort algorithm requires exactly $n(n - 1)/2$ comparisons but we don't know how many exchange steps we need. The number of exchange steps depends on the input sequence. For example, if the input sequence is <1, 2, . . ., n>, no exchange steps are needed. If the input sequence is <n, $n - 1$, . . ., 1>, the algorithm performs $n(n - 1)/2$ exchange steps.

Furthermore, the running time of each comparison or exchange step depends on the physical device which executes this algorithm. So, the exactly running time depends on many factors and is difficult to estimated. However, we can always assume the running time of one step (either comparison or exchange) is bounded by some constant which is independent of n for any physical device.

By using the asymptotic notations, we can safely say that the running time of the bubble sort algorithm is $O(n^2)$. On the other hand, it is also $\Omega(n^2)$ since we need $n(n - 1)/2$ comparison steps. Thus, the time complexity of the bubble sort algorithm is $\Theta(n^2)$.

The asymptotic notations help us ignore some details of the running process of the algorithm and focus on the dominant term in the running time. It shows that, when the input size grows, the running time of bubble sort algorithm grows quadratically, but neither linearly nor exponentially.

4.2 Divide-and-Conquer Algorithms

Divide-and-conquer is an algorithm design paradigm based on the idea that we recursively break down a problem into two or more subproblems of the similar type, until these subproblems become simple enough to be solved directly. In this section, we will use several examples to illustrate the idea of divide and conquer method. We firstly focus on how different division methods affect the performance of the algorithm. Sections 4.2.1 and 4.2.2 consider the sort problem again. They provide two different ways to split the original problem into subproblems, and show different performance correspondingly. In Sect. 4.2.3, we show an interesting example which illustrates that equal division is not always the best idea. Then, in Sects. 4.2.4 and 4.2.5, we will learn how to efficiently combine the results of subproblems so as to obtain the result of the original problem. These two examples illustrate that the construction of the subproblems and the combination process should closely bound together. Finally, in Sect. 4.2.6, we summarize the key points in the divide-and-conquer methodology.

4.2.1 The Insertion Sort Algorithm

In the sorting problem, we want to order the sequence with n integers from small to large. One natural idea is to firstly sort the first $n - 1$ integers and then insert the n-th integer into the proper position of a sorted sequence. How can we sort the first $n - 1$ integers? Well, this is a subproblem with smaller size. This leads to the idea of the insertion sort algorithm (Fig. 4.2).

In each round, the first i integers of the sequence are already in order. Our task is to insert the $(i + 1)$-th integer into the proper position. Table 4.3 shows the detailed process for the example input [6, 2, 4, 1, 5, 9].

The insertion sort algorithm is not a typical example of divide-and-conquer method. But it illustrates the idea of subproblem. For the sequence of n unsorted integers, we divide it into two subproblems: one with the first $n - 1$ integers, and the

```
Input: An array A of length n to be sorted, e.g., A=[6, 2, 4, 1, 5, 9]
Output: A sorted array A, e.g., A=[1, 2, 4, 5, 6, 9].
Steps:
   for i = 1 to n-1
         j=i+1;
         while (j>1) and (A[j-1]>A[j])
              exchange A[j] with A[j-1];
              j=j-1;
```

Fig. 4.2 The insertion sort algorithm

Table 4.3 The sequence of steps for insertion sorting [6, 2, 4, 1, 5, 9] into [1, 2, 4, 5, 6, 9]

Outer loop	Inner loop	State before	State after
First round	1st comparison 6>2, exchange	6, 2	2, 6
Second round	1st comparison 6>4, exchange	2, 6, 4	2, 4, 6
	2nd comparison 2<4, no exchange	2, 4, 6	2, 4, 6
Third round	1st comparison 6>1, exchange	2, 4, 6, 1	2, 4, 1, 6
	2nd comparison 4>1, exchange	2, 4, 1, 6	2, 1, 4, 6
	3rd comparison 2>1, exchange	2, 1, 4, 6	1, 2, 4, 6
Fourth round	1st comparison 6>5, exchange	1, 2, 4, 6, 5	1, 2, 4, 5, 6
	2nd comparison 4<5, no exchange	1, 2, 4, 5, 6	1, 2, 4, 5, 6
Fifth round	1st comparison 6<9, no exchange	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9

```
InsertionSort(n)   //sort sequence A[1] to A[n]
   **if** (n==1) **then** return;
   InsertionSort(n-1);        //solve the sub-problem with A[1] to A[n-1]
    j=n;
    **while** (j>1) and (A[j-1]>A[j])
           exchange A[j] with A[j-1];
           j=j-1;
```

Fig. 4.3 The insertion sort algorithm (revision)

```
MergeSort([A[1],…,A[n]])   //sort sequence A[1] to A[n]
   **if** (n==1) **then** return [A[1]];
   B=MergeSort([A[1],…,A[n/2]]);
   C=MergeSort([A[n/2+1],…,A[n]]);
   return merge(B, C);
```

Fig. 4.4 The merge sort algorithm

other with the last integer. Suppose we can solve two subproblems while the second one is trivial; we only need to insert the last integer into the sorted sequence.

We can revise the insertion sort algorithm in the following way to emphasize the idea of subproblems (Fig. 4.3).

Now let us consider the time complexity of insertion sort algorithm. Let $T(n)$ denote the time complexity of the insertion sort algorithm for n unsorted integers. We have

$$\begin{aligned} T(1) &= 0; \\ T(n) &= T(n-1) + \text{time for insertion} \\ &\leq T(n-1) + cn \end{aligned}$$

for some constant c. Thus, we have $T(n) = O(n^2)$.

4.2.2 The Merge Sort Algorithm

In the insertion sort algorithm just discussed above, we divide the original problem into two unequal subproblems, where one subproblem contains $n - 1$ integers and the other subproblem contains only one integer.

Figure 4.4 shows another sorting algorithm called the merge sort algorithm. Here, we divide the original problem into two subproblems, which deal with almost equal number of integers. That is, the sorting problem of MergeSort([A[1],. . .,A[n]]) is first divided into two subproblems: MergeSort([A[1],. . .,A[n/2]]) and MergeSort([A [n/2+1],. . .,A[n]]). Then we merge the results B and C of the two subproblems, which are each a sorted sequence of integers.

```
merge(B, C)   // merge two sorted sequences
   while (B is not empty) and (C is not empty)
        b = first integer in B;
        c = first integer in C;
        if (b<c) then
             append A with b;
             delete b from B;
        else
             append A with c;
             delete c from C;
   while (B is not empty)
        b = first integer in B;
        append A with b;
        delete b from B;
   while (C is not empty)
        c = first integer in C;
        append A with c;
        delete c from C;
   return A;
```

Fig. 4.5 The merge function in the merge sort algorithm

Table 4.4 The sequence of steps for the merge function

Comparison	State before	State after
1st comparison 2>1, delete 1 from C	A: B: 2, 4, 6 C: 1, 5, 9	A: 1 B: 2, 4, 6 C: 5, 9
2nd comparison 2<5, delete 2 from B	A: 1 B: 2, 4, 6 C: 5, 9	A: 1, 2 B: 4, 6 C: 5, 9
3rd comparison 4<5, delete 4 from B	A: 1, 2 B: 4, 6 C: 5, 9	A: 1, 2, 4 B: 6 C: 5, 9
4th comparison 6>5, delete 5 from C	A: 1, 2, 4 B: 6 C: 5, 9	A: 1, 2, 4, 5 B: 6 C: 9
5th comparison 6<9, delete 6 from B	A: 1, 2, 4, 5 B: 6 C: 9	A: 1, 2, 4, 5, 6 B: C: 9
no comparison delete 9 from C	A: 1, 2, 4, 5, 6 B: C: 9	A: 1, 2, 4, 5, 6, 9 B: C:

How can we merge two integer sequences B and C? If B and C are two arbitrary sequences of integers, the merging problem is as difficult as the original sorting problem. But remember that, we already know an important fact: B and C are two sorted sequences of integers.

Figure 4.5 shows one of the ways to merge two sorted sequences. Table 4.4 shows the example process for merging two sorted sequences B=[2, 4, 6] and C=[1, 5, 9].

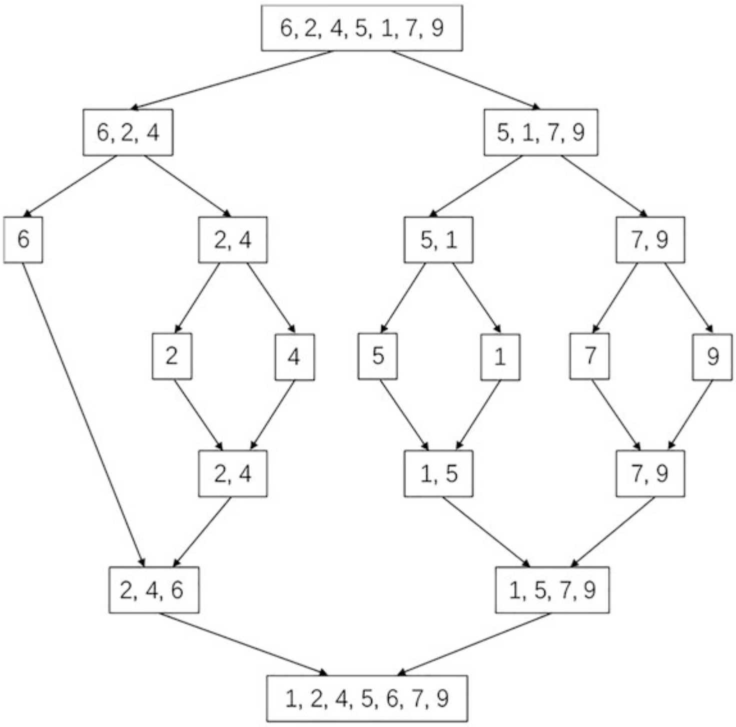


Fig. 4.6 The process of merge sort algorithm with input [6, 2, 4, 5, 1, 7, 9]

We are now ready to discuss the main part of the merge sort algorithm in details. Note that when we write down B = MergeSort([A[1],. . .,A[n/2]]), we recursively call the merge sort algorithm for a smaller subproblem with input sequence A[1],. . ., A[n/2]. In this subproblem, we will again divide it into 2 sub-subproblems: A[1],. . ., A[n/4] and A[n/4+1],. . .,A[n/2], and recursively solve the sub-subproblems with the merge sort algorithm. The recursion will end if the size of unsorted sequence is 1 which is the trivial case. Figure 4.6 illustrates an example for the input sequence [6, 2, 4, 5, 1, 7, 9].

Finally, let us consider the time complexity of the merge sort algorithm. Let $T(n)$ denote the time complexity for n unsorted integers. Similar to the insertion sort algorithm, we have

$$T(1) = 0;$$

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \text{time for merge function}$$

$$\leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn$$

for some constant c. Thus, we have $T(n) = O(n \log n)$ which is more efficient than the insertion sort algorithm.

The framework of both the insertion sort algorithm and the merge sort algorithm is the same. The cost of "insertion of the last element" in the insertion sort algorithm and the cost of "merge function" in the merge sort algorithm are also roughly the same, where each requires at most $n - 1$ comparisons. The main difference is the sizes of the two subproblems. In the insertion sort algorithm, the sizes are 1 vs. $n - 1$, while in the merge sort algorithm, the sizes are $n/2$ vs. $n/2$. This reduces the time complexity from $O(n^2)$ to $O(n \log n)$ for the sorting problem. It illustrates that when we want to use the divide-and-conquer method to solve a problem, it is important to smartly divide the original problem into subproblems. Often, it is smart to divide a problem into two subproblems of almost equal sizes.

4.2.3 Single Factor Optimization

In this section, we will discuss an interesting problem called single factor optimization. One illusion of the people who just learn the divide-and-conquer method is that they may blindly believe the power of equal division, like the one we did in the merge sort algorithm. This section illustrates that it is not always the case.

The single factor optimization problem considers a univariate function f defined in the interval [a, b]. Assume that f satisfies the following single-peak condition: f firstly (strictly) monotonically increases and then (strictly) monotonically decreases. How can we quickly find the point x that maximize $f(x)$?

Well, if function f has good properties, we might compute the maximum directly. For example, if we know the explicit representation of the function f, we can calculate the zero point of its derivative. But in this section, we assume f is implicitly accessed by an oracle such that the only allowed operation is that given x, the oracle will return the value $f(x)$. Our goal is to minimize the number of oracle queries.

Generally, in order to facilitate computer processing, we need to transfer the problem into a discrete version. Suppose we discretize the interval $[a, b]$ into n points, and the function f is represented by an array A: A[1], A[2],..., A[n]. The choice of n depends on the precision we want to achieve. In this way, the problem can be described as the following searching problem:

The Single Factor Optimization Problem

- **Input**: array A[1], A[2], ..., A[n] such that $\exists i$, $1 \leq i \leq n$, A[1]<A[2]<$\cdots$<A[i], and A[i]>A[i+1]>$\cdots$>A[n].
- **Output**: i and A[i].

```
Algorithm 1: find the maximum based on equal division
    Input: A[1], A[2],…, A[n] which satisfies single-peak property
    Output: the maximum value in the array A[i], i

    begin=1; end=n;
    While (end-begin>1) do
            mid = (begin+end)/2;
        If (A[mid]<A[mid+1]) then
                begin=mid+1;
        Else
                end=mid;
    If (A[begin]<A[end]) then
        Return A[end], end
    Else
        Return A[begin], begin
```

Fig. 4.7 The single-factor optimization (binary search)

The simplest way to solve this problem is to query the array A one by one. The worst case needs n queries. This method can be used to find the maximum value of any array and obliviously does not take full advantage of the "single-peak" property.

A natural idea is to search from the middle. We select to query $A\left[\frac{n}{2}\right]$ and $A\left[\frac{n}{2}+1\right]$. There are several cases:

1. If $A\left[\frac{n}{2}\right] > A\left[\frac{n}{2}+1\right]$, then according to the property of the function, we know that the maximum value of the function is in the interval $\left[1, \frac{n}{2}\right]$, so we can discard the interval $\left[\frac{n}{2}+1, n\right]$;
2. If $A\left[\frac{n}{2}\right] < A\left[\frac{n}{2}+1\right]$, similar to 1), we can determine that the maximum value of the function is in the interval $\left[\frac{n}{2}+1, n\right]$, so we can discard the interval $\left[1, \frac{n}{2}\right]$;
3. The case $A\left[\frac{n}{2}\right] = A\left[\frac{n}{2}+1\right]$ is impossible.

In either case, we have reduced the search interval from $[1, n]$ by half. In the new search interval, which could be $\left[1, \frac{n}{2}\right]$, or $\left[\frac{n}{2}+1, n\right]$, the function f still satisfies the property of single-peak condition. We can recursively call this algorithm to continue searching for the maximum point of the function.

The algorithm we described above can be formalized into the algorithm shown in Fig. 4.7.

Let's take a look at the efficiency of this algorithm. We use $T(n)$ to denote the number of queries required by an algorithm on an input of length n. In the first step of the algorithm, we need to query twice: the function values $f\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$,$f\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)$. In the second step of the algorithm, we reduce the problem with the original input size n to an input size of $\left\lfloor\frac{n}{2}\right\rfloor$ or $(n-\left\lfloor\frac{n}{2}\right\rfloor)$ subproblem. The setting of this subproblem is exactly the same as the original problem, except that the scale is smaller than the original problem. Therefore, if the algorithm is called recursively, the required number of queries is $T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$ or $T\left(n-\left\lfloor\frac{n}{2}\right\rfloor\right)$. Combining these two steps, we can get:

```
Algorithm 2: find the maximum based on the golden section method
Input: $A[1], A[2], \cdots, A[n]$
Output: the maximum value in the array
Steps:
 $begin \leftarrow 1,\ end \leftarrow n$
 While $end - begin > 1$ do
         $x_1 \leftarrow \lfloor \alpha \cdot begin + (1-\alpha) \cdot end \rfloor, x_2 \leftarrow \lfloor (1-\alpha) \cdot begin + \alpha \cdot end \rfloor$
         If $A[x_1] \le A[x_2]$ then
               $begin \leftarrow x_1$
         Else
               $end \leftarrow x_2$
 If $A[begin] < A[end]$ then
         Return $A[end], end$
 Else
         Return $A[begin], begin$
```

Fig. 4.8 The single-factor optimization (golden section method)

$$T(n) \le \max \left\{ T\left(\left\lfloor \frac{n}{2} \right\rfloor \right), T\left(n - \left\lfloor \frac{n}{2} \right\rfloor \right) \right\} + 2 \le T\left(\left\lfloor \frac{n+1}{2} \right\rfloor \right) + 2$$

In addition, we know that the initial value T (1) = 1. Thus, we have

$$T(n) \le 2\lceil \log n \rceil + 1$$

In the above algorithm, we make two queries each time and reduce the length of the interval by half. Is it possible to further improve this algorithm? It seems to be the most economical to shrink the interval by half each time, because if we divide the interval into two parts, the length of one part will always be at least the half. Can we reduce the number of queries in each round? This is possible, if we can reuse the query results which we have obtained before.

Below we give another more efficient algorithm for the above problem. The idea of Algorithm 2 is basically the same as that of Algorithm 1. The key difference lies in the selection of cut points. In our algorithm and analysis, the constant $\left(\sqrt{5}-1\right)/2 \approx 0.618$ which is actually the golden section ratio is frequently used. For the convenience of writing, we set $\alpha = \left(\sqrt{5}-1\right)/2$. The block diagram of Algorithm 2 is shown in Fig. 4.8:

Now, let us analyze the performance of Algorithm 2. In order to simplify the analysis, we ignore all rounding symbols. By querying $x_1 = (1-\alpha)\,n$ and $x_2 = \alpha n$, we reduce the problem to a subproblem $A[1, \ldots, \alpha n]$ or $A[(1-\alpha)\,n, \ldots, n]$. In either case, the scale of the new subproblem is αn. It seems that it is worse than binary search. However, notice that for the new subproblem, one of the two points we need to query is already known! Taking $A[1, \ldots, \alpha n]$ as an example, according to the steps of the algorithm, the two points we need to query are $y_1 = \alpha(1-\alpha)n$ and $y_2 = \alpha^2 n$. Note that α is the golden section ratio which is the solution of the equation $x^2 + x - 1 = 0$. After simple calculation, we have $y_2 = x_1$, which means that we do not need to query the value of point y_2 because we already know the value of point x_1. Similarly, for the subproblem $A[(1 - \alpha)\, n, \ldots, n]$, we can know that the first branch point required by the algorithm is exactly x_2. In either case, we only need to query ONE new point, so we get the following recursion:

$$\begin{cases} \mathrm{T}(n) = \mathrm{T}(\alpha n) + 1, \\ \qquad\quad \mathrm{T}(1) = 1. \end{cases}$$

By solving this recursion, we have

$$\mathrm{T}(n) = \log_{(1+\alpha)} n + 1$$

Comparing the two algorithms, we can see that the performance of Algorithm 2 is better than that of Algorithm 1 (because $2\log_2 n = \log_{\sqrt{2}} n > \log_{(1+\alpha)} n$). To solve the same problem, when we use different methods to design our algorithm, its performance is different. We always hope to be able to design the best algorithm, that is, the most efficient algorithm to solve the problem.

In single-factor optimization, our intuition is to reduce the query number in each round. We come up with the brilliant idea that we can reuse the query in the previous round. In order to use such an idea, we modify the division method by using golden section. This tells us: the division method and the combination method are interdependent, and no division method is universal.

There is a final remark of the single-factor optimization problem. In all of our discussion, our objection is to minimize the number of oracle queries. Since in most real scenarios, one oracle query is much more expensive than the comparison operations or assignment operations in the algorithm, it is natural to ignore the cost of the other operations. But, if the running time of one oracle query is the same as that of one comparison operation or one assignment operation, is algorithm 2 still better than algorithm 1?

4.2.4 Integer Multiplication

In this section, we will discuss the integer multiplication problem. We will show if we apply divide-and-conquer method mechanically, we will not enhance performance. Thus, we need to think about a cleverer way to solve the problem.

Firstly, let us describe the integer multiplication problem:

The Integer Multiplication Problem

- Input: $X = x_n x_{n-1} \ldots x_1$, $Y = y_n y_{n-1} \ldots y_1$.
- Output: $Z = X \times Y = XY$

Calculating the multiplication of two numbers is a problem we often encounter in our daily life. We give an example to show how to calculate the multiplication of two 3-digit numbers 123 $\times$ 321 by hand.

$$\begin{array}{r}
123\\
\times 321\\
\hline
123\\
246\\
369\\
\hline
39483
\end{array}$$

For two n-digit numbers (imagine that n is very large, for example, $n = 10^{12}$), if we use a similar method to calculate the multiplication, we need n^2 multiplications and about n^2 additions of 1-digit operation. Let us see if we can reduce the total number of calculations by adopting the divide-and-conquer approach.

Write the input X and Y as follows:

$$X = X_1 \times 10^{n/2} + X_2, Y = Y_1 \times 10^{n/2} + Y_2,$$

where the length of X_1, X_2, Y_1, Y_2 is $n/2$. What we need to calculate is:

$$Z = XY = X_1Y_1 \times 10^n + (X_1Y_2 + X_2Y_1) \times 10^{n/2} + X_2Y_2.$$

The naïve idea is to call the algorithm recursively to calculate X_1Y_1, X_1Y_2, X_2Y_1, X_2Y_2. Based on this idea, we need to multiply two (n/2)-digits numbers 4 times in total, and also need up to 3 times n-digits addition, so we get

$$\begin{cases} \mathrm{T(n)} = 4\mathrm{T}(n/2) + 3n, \\ \qquad\qquad \mathrm{T}(1) = 1. \end{cases}$$

It is easy to solve the recursion and obtain $\mathrm{T}(n) = \mathrm{O}(n^2)$, where there is no substantial improvement over the previous natural algorithm.

Now let us change the way of thinking. The idea is: what we need is $X_1Y_2 + X_2Y_1$ instead of X_1Y_2 and X_2Y_1.

Notice that $X_1Y_1 + X_1Y_2 + X_2Y_1 + X_2Y_2 = (X_1 + X_2)(Y_1 + Y_2)$. So, by calculating X_1Y_1, X_2Y_2, $(X_1 + X_2)(Y_1 + Y_2)$, and then use

$$X_1Y_2 + X_2Y_1 = (X_1 + X_2)(Y_1 + Y_2) - X_1Y_1 - X_2Y_2,$$

we can obtain $X_1Y_2 + X_2Y_1$. By this method, we need to multiply two (n/2)-digits numbers 3 times in total, and also need n-digits addition 6 times: $X_1 + X_2$, $Y_1 + Y_2$, $(X_1 + X_2)(Y_1 + Y_2) - X_1Y_1 - X_2Y_2$, and

$$Z = X_1Y_1 \times 10^n + (X_1Y_2 + X_2Y_1) \times 10^{n/2} + X_2Y_2.$$

Thus, we have

$$\begin{cases} \mathrm{T}(n) = 3\mathrm{T}(n/2) + 6n, \\ \mathrm{T}(1) = 1. \end{cases}$$

By solving this recursion, we have $\mathrm{T}(n) = cn^{\log_2 3} + O(n) \approx n^{1.59}$, which is a great improvement compared to the naïve algorithm with time complexity $O(n^2)$. This example tells us when designing divide-and-conquer algorithms, it is important to make the number of subproblems of recursive calls as small as possible.

4.2.5 Matrix Multiplication

Matrix multiplication is a natural extension of integer multiplication. We want to further illustrate the idea on how to minimize the number of subproblems.

The Matrix Multiplication Problem

- **Input**: two $n \times n$ matrices $\mathrm{A} = [a_{i,\,j}]$, $\mathrm{B} = [b_{i,\,j}]$.
- Output: C = AB.

According to the definition of matrix multiplication, we know

$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \ldots + a_{i,n}b_{n,j}.$$

If we use the natural method to compute each $c_{i,\,j}$ directly, we need $\mathrm{O}(n^3)$ multiplications and $\mathrm{O}(n^3)$ additions in total. Let use divide A, B and C into four $(n/2 \times n/2)$ sub-matrices:

$$A = \begin{bmatrix} \mathrm{A}_{1,1} & \mathrm{A}_{1,2} \\ \mathrm{A}_{2,1} & \mathrm{A}_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} \mathrm{B}_{1,1} & \mathrm{B}_{1,2} \\ \mathrm{B}_{2,1} & \mathrm{B}_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} \mathrm{C}_{1,1} & \mathrm{C}_{1,2} \\ \mathrm{C}_{2,1} & \mathrm{C}_{2,2} \end{bmatrix}.$$

Then, we have

$$\mathrm{C}_{1,1} = \mathrm{A}_{1,1}\mathrm{B}_{1,1} + \mathrm{A}_{1,2}\mathrm{B}_{2,1}$$
$$\mathrm{C}_{1,2} = \mathrm{A}_{1,1}\mathrm{B}_{1,2} + \mathrm{A}_{1,2}\mathrm{B}_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

If we directly call the subroutine to compute $C_{1,\ 1}$, $C_{1,\ 2}$, $C_{2,\ 1}$, $C_{2,\ 2}$, we need 8 calls of the subproblem of the multiplication of $n/2 \times n/2$ submatrices. In addition, we need 4 times addition of $n/2 \times n/2$ matrices. Thus, the recursion is

$$T(n) = 8T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2$$

The final complexity obtained by solving this recursion is still $O(n^3)$. Applying the previous ideas about n-digit multiplication, we need to reduce the number of subroutine calls through appropriate addition and subtraction. How can we achieve this? It is more difficult than the integer multiplication problem. The following solution is proposed by Volker Strassen (1936-).

Example 4.4. Strassen's Algorithm for Matrix Multiplication
Define the following 7 matrices:

$$\begin{aligned}
M_1 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}),\\
M_2 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}),\\
M_3 &= (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2}),\\
M_4 &= (A_{1,1} + A_{1,2})B_{2,2},\\
M_5 &= A_{1,1}(B_{1,2} - B_{2,2}),\\
M_6 &= A_{2,2}(B_{2,1} - B_{1,1}),\\
M_7 &= (A_{2,1} + A_{2,2})B_{1,1}.
\end{aligned}$$

We can make an observation: the matrices we need to compute, e.g., $C_{1,\ 1}$, $C_{1,\ 2}$, $C_{2,\ 1}$, $C_{2,\ 2}$, can be calculated by using $M_1, \ldots, M_7$ and some addition or subtraction operations. The detailed method is as follows:

$$\begin{aligned}
C_{1,1} &= M_1 + M_2 - M_4 + M_6,\\
C_{1,2} &= M_4 + M_5,\\
C_{2,1} &= M_6 + M_7,\\
C_{2,2} &= M_2 - M_3 + M_5 - M_7.
\end{aligned}$$

Let's take a look at the performance of Strassen's algorithm. First of all, Strassen's algorithm needs to call a total of seven sub-matrix multiplications. In addition, the algorithm also needs to add $n/2 \times n/2$ matrices several times, so we have the following recursion:

$$\mathrm{T}(n) = 7\mathrm{T}\left(\frac{n}{2}\right) + O\left(n^2\right).$$

Here we do not accurately calculate the total number of times required for the addition or subtraction. Instead, we use the $O(\cdot)$ notation to hide the constant. In fact, this constant does not affect the magnitude of T(n). No matter what the constant is, the solution is $\mathrm{T}(n) = \mathrm{O}(n^{\log 7}) \approx \mathrm{O}(n^{2.81})$.

The algorithm proposed by Strassen in 1969 is the first algorithm about matrix multiplication that can beat the conventional $\mathrm{O}(n^3)$ algorithm. Since then, the upper bound of the complexity of matrix multiplication has been continuously improved: the algorithm complexity proposed by Pan in 1978 is $\mathrm{O}(n^{2.796})$, by Bini et al. in 1979 is $\mathrm{O}(n^{2.78})$, by Schönhage in 1981 is $\mathrm{O}(n^{2.548})$, by Romani in 1982 is $\mathrm{O}(n^{2.517})$, by Strassen in 1986 is $\mathrm{O}(n^{2.479})$. At present, the best matrix multiplication algorithm was proposed by Coppersmith and Winograd in 1987. The complexity of the algorithm was $\mathrm{O}(n^{2.376})$ when originally proposed. Recently, the analysis of the original algorithm has been continuously improved by Stothers, Williams, Le Gall and others. The algorithm complexity is reduced to $\mathrm{O}(n^{2.3729})$. Whether there is a matrix multiplication algorithm close to $\mathrm{O}(n^2)$ complexity is an important unsolved problem in the field of algorithms.

4.2.6 Summarization

In this section, we discuss many examples with the help of divide-and-conquer methodology. In general, divide-and-conquer comes from the idea that when you want to solve a complicated problem, try to transfer it into some easier problem. There are two features of the problems which can be solved with the help of divide-and-conquer method. Firstly, the problem with extremely small size is straightforward to solve, for example, the sort problem with only two elements. This will be served as the basis of the recursive process. Secondly, we can solve the general problem with the help of the problem with smaller size. This part is the art in the divide-and-conquer method. There is no universal way of construction for every scenario, and we need to observe the specialty for each problem ourselves. However, there are two things which are usually important in the design of divide-and-conquer algorithms.

Firstly, it is usually better to use smaller number of subproblems to solve the original problem. In many examples, such as integer multiplication and matrix multiplication, we are trying to reduce the number of subproblems and we show with smaller number of subproblems, we dramatically improve the performance of the algorithms, even if we slightly increase the time complexity for each round. But please do not go to the other extreme. For example, in the integer multiplication problem, the natural idea gives us $\mathrm{T(n)} = 4\mathrm{T}(n/2) + 3n$. If we modify it into T

(n) = 3T(n/2) + 6n as illustrated in Sect. 4.2.4, it improves the performance. But if we modify it into T(n) = 3T(n/2) + $O(n^2)$, the performance will be T(n) = O(n^2). If T (n) = 3T(n/2) + O(n^3), the performance will be even worse than the natural algorithm. In other words, in the design of divide-and-conquer algorithms, we need to balance all parts.

Secondly, it is usually better to partition the original problem into subproblems. For example, in the sort algorithm, we partition the whole unsorted set into two disjoint subsets and recursively sort them. It makes no sense if the subproblems have overlapping elements. Partition, in some sense, can make the size of subproblems as small as possible. Thus, it is useful for better performance. But in the example of single factor optimization, we also see that this is not always the case. In such an example, although two possible subproblems $[1, \ldots, \alpha n]$ and $[(1 - \alpha)n, n]$ are overlapping, it is faster than the natural halving method. In the integer multiplication and the matrix multiplication problems, it is even hard to distinguish which idea is a partition. When dividing the problem into subproblems, the partition method is usually a good start point since it makes the size of subproblems small, but the size of subproblems is not the only factor in the divide-and-conquer method, and we need to balance all parts. On the other hand, if there are too many overlaps between different subproblems, some other methods may be more powerful than divide-and-conquer. See Sect. 4.3.1 for an example.

4.3 Other Examples of Interesting Algorithms

In the previous section, we focused on the algorithms based on the divide-and-conquer method. We also learn how to analyze the algorithm complexity through recursion expression. In this section we will see some other examples of algorithms. The first example is using dynamic programming to compute Fibonacci number. We will see how this method can eliminate duplicated computation. The second example is the stable matching problem, which uses a kind of "greedy" algorithm method, different from the divide-and-conquer paradigm. While the correctness of a divide-and-conquer algorithm is usually straightforward, the correctness of a greedy solution needs to be proved. The final example is the quicksort algorithm for the sorting problem. The quicksort algorithm is not a deterministic algorithm, that is, the algorithm will toss some coins to decide the next step during the process of running. We will see how to analyze the complexity of such algorithms.

4.3.1 Dynamic Programming

A divide-and-conquer algorithm solves a problem by dividing the problem into *independent* subproblems, and then combining their solutions. The key character here is that the subproblems are independent, meaning that they usually do not overlap. That is, the subproblems do not share subproblems and do not solve the shared subproblems multiple times.

What if the subproblems do overlap? An algorithm paradigm called dynamic programming specifically addresses such concerns. A dynamic programming algorithm divides the problem into potentially overlapping subproblems and combines their solutions. The key character is that the solution to every shared subproblem is memorized and reused, avoiding computing its solution multiple times.

There are two approaches to implementing memorization in dynamic programming algorithms, the top-down approach and the bottom-up approach. We analyze the example of computing a small Fibonacci number F(5) to see how the two approaches in dynamic programming work, as shown in Example 4.5.

Example 4.5. Eliminate Redundant Computation by Dynamic Programing
We analyze the behaviors of two dynamic programming algorithms computing a small Fibonacci number F(5), against the recursive program fib-5.go. The details of the fib-5.go program and the fib.dp-5.go program using the top-down approach are shown in Fig. 4.9. Some diagnostic print statements are added to print out intermediate results, to reveal the behaviors of these programs.

The recursive program fib-5.go does a lot of redundant, unnecessary computations. This becomes immediately clear when we look at Fig. 4.10, which shows the tree of recursive calls to fibonacci(n), denoted as F(5), (F4), F(3), F(2), F(1), and F (0). The circled numbers, ①, ②, . . ., ⑮, show the order as to how the calls are made. The program calls fibonacci(n) 15 times. It first calls F(5), then F(4), and finally F(0). Note that F(0) is called 3 times, F(1) 5 times, F(2) 3 times, F(3) 2 times. Altogether, 9 unnecessary computations are performed.

The fib.dp-5.go program uses the top-down approach of dynamic programming to compute F(5). It is similar to the recursive program fib-5.go, but results of F(n) are stored in a 6-element array mem. Every element mem[i] is initialized to -1, to denote that this element has not been computed yet. When the program calls fibonacci(n), the code first checks if mem[n] is -1. If it is not, the function call immediately returns with the already computed value mem[n], without going further to do unnecessary computation.

The diagnostic printout should show that when running the fib.dp-5.go program, fibonacci(n) is called only 9 times. The calling order is F(5), (F4), F(3), F(2), F(1), F (0), F(1), F(2), F(3). Furthermore, the last three calls F(1), F(2), F(3) are returned immediately, without doing unnecessary computation. Thus, the fib.dp-5.go program only performs 6 necessary computations F(5), (F4), F(3), F(2), F(1), F(0).

To compute Fibonacci number F(n), it is easy to find that we need to call F(i) for all $i < n$. The bottom-up approach takes advantage of this fact and prepares the small Fibonacci number before a call. It starts at the smallest subproblems F(0) and F(1), memorize their solutions, and combines their solutions into the solution of the subproblem next level up, e.g., F(2)=F(1)+F(0). This iterative process continues, to obtain F(3)=F(2)+F(1), F(4)=F(3)+F(2), until the topmost solution F(5) is obtained. When we compute some Fibonacci number, for example F(3), we do not need to worry whether the smaller numbers F(2) and F(1) have been computed or

```
package main
import "fmt"
func main() {
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  fmt.Println("F(",n,")")
  if n == 0 || n == 1 {
    return n
  }
  return fibonacci(n-1)+fibonacci(n-2)
}
```

(a)

```
package main
import "fmt"
var mem [6]int
func main() {
        for i := 0; i < 6; i++ { mem[i] = -1 }
        fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
        fmt.Println("F(",n,")")
        if mem[n] != -1 {
                fmt.Println("Immediate Return: F(",n,")=",mem[n])
                return mem[n]
        }
        if n == 0 || n == 1 {
                mem[n] = n
                fmt.Println("Return: F(",n,")=",mem[n])
                return mem[n]
        }
        mem[n] = fibonacci(n-1) + fibonacci(n-2)
        fmt.Println("Return: F(",n,")=",mem[n])
        return mem[n]
}
```

(b)

Fig. 4.9 Two programs to compute Fibonacci number F(5). (**a**) Recursive program fib-5.go. (**b**) Dynamic programming program fib.dp-5.go

not. Thus, we do not need -1 to represent unfinished work, as what we did in fib.dp-5.go. The bottom-up program fib.dp.bu.go is shown below. In this code, we do not

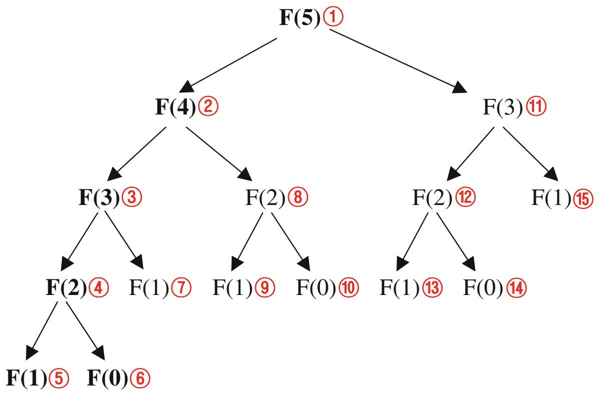


Fig. 4.10 The sequence of fibonacci(n) calls, where ①, ②, . . . denote the order

even store all numbers we compute, since we only need F(n-1) and F(n-2) to computer F(n).

```
package main   // program fib.dp.bu.go
import "fmt"
func main() {
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  a :=0
  b :=1
  for i :=1; i < n+1; i++ {
    a = a + b
    a, b = b, a
  }
  return a
}
```

The bottom-up approach often results in a simpler, iterative program. However, students may find either the top-down or the bottom-up approach more intuitive and easier to use. For instance, for the problem of finding a shortest path in a graph, many students prefer the top-down approach.

4.3.2 (***) The Greedy Strategy

This example comes from economics. The 2012 Nobel Prize in Economics was awarded to mathematical economists Alvin Roth and Lloyd Shapley in recognition of their outstanding contributions to “the theory of stable distribution and its market

W_1	M_2	M_1	M_3
W_2	M_1	M_2	M_3
W_3	M_2	M_3	M_1

M_1	W_2	W_1	W_3
M_2	W_1	W_2	W_3
M_3	W_3	W_1	W_2

Fig. 4.11 Matrix W and matrix M in an example of the stable matching problem

design practice". The stable matching problem is one of the starting point of this research area.

Consider the following scenario. Suppose n boys $M_1, M_2, \ldots, M_n$ and n girls $W_1, W_2, \ldots, W_n$ participate in a dance together, and each of them hopes to find a suitable partner to dance. For each girl W_i, according to her own criteria, there is a ranking for the n boys. The boy in the higher rank indicates that W_i thinks he is more suitable than the boy in the lower rank. Similarly, for each boy M_j, there will also be a ranking for all n girls. Suppose that at the beginning of the dance, they arbitrarily formed n pairs of dance partners and began to dance the first dance. In this process, if there exists a pair of boys M_i and girls W_j, they are not each other 's partners, but each of them feels that the other one is better than their current partner, then when the next song starts, they will choose the other as their partner instead of the current partner. We call such a pair (girl, boy) an unstable pair. If there is an unstable pair in the matching, we call such a matching unstable, otherwise we call such a matching stable. The question now is whether these n girls and these n boys can form n pairs of stable partners together.

This problem is called the **stable matching problem**. Below we give a more rigorous mathematical description of this problem:

We can use two $n \times n$ matrices to represent our input. Matrix W represents the preference matrix for girls. Each row is a permutation of $\{1, 2, \ldots, n\}$, and the i-th row means that the ranking of boys for the girl W_i. Matrix M represents the preference matrix of boys, and the i-th row represents the ranking of M_i for all girls. An example is given in the table in Fig. 4.11.

The stable matching problem:

Is there a matching between boys and girls $\left\{\left(W_{i_1}, M_{j_1}\right), \left(W_{i_2}, M_{j_2}\right), \ldots, \left(W_{i_n}, M_{j_n}\right)\right\}$, where $\{i_1, \ldots, i_n\}$ and $\{j_1, \ldots, j_n\}$ are two permutations of $\{1, 2, \ldots, n\}$, satisfying that there is no pair $\left(W_{i_k}, M_{j_\ell}\right)$, where $k \neq \ell$, such that M_{j_ℓ} ranks higher than M_{j_k} in the order of W_{i_k}, and W_{i_k} ranks higher than W_{i_ℓ} in the order of M_{j_ℓ}.

For example, in the example in Fig. 4.12, $\{(W_1, M_1), (W_2, M_2), (W_3, M_3)\}$ is an unstable matching. Let us examine girl W_1 and boy M_2. W_1's current partner is M_1, while in the ranking of W_1, M_2 ranks higher than M_1. At the same time, for the boy M_2's current partner W_2, W_1 is also ranked higher than W_2. So (W_1, M_2) forms an unstable pair. It can be verified that the matching $\{(W_1, M_2), (W_2, M_1), (W_3, M_3)\}$ is a stable matching. Note that for unstable boy and girl pairs, both parties must feel that the other is better. If only one party feels that the other is better, this does not

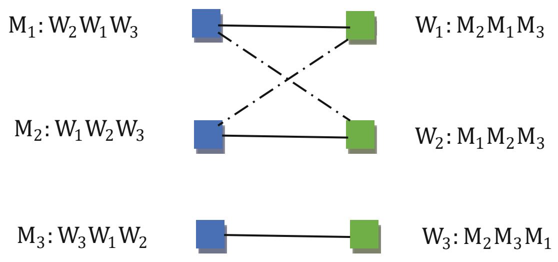


Fig. 4.12 Example of stable matching and unstable matching

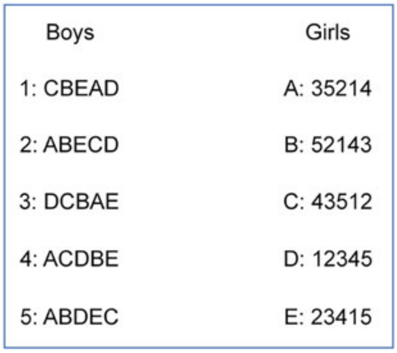
constitute an unstable pair. For example, consider the pair M_2 and W_3 here. Although according to the order of W_3, M_2 is better than its current partner for M_3, in the view of M_2, his partner W_2 is better than W_3. Thus he will not agree to change the partner.

This problem has important applications in economics. Mathematical economists Gale and Shapley first proposed and studied this problem. They proved that regardless of the preference ranking of each boy and girl, a stable matching always exists. In fact, they have given an algorithm to find such a matching. This algorithm is called the Gale-Shapley algorithm today, described as follows.

The algorithm is divided into several rounds. In the first round, each boy selects the girl who has the highest ranking in his preference order and invites the girl to dance. For any girl *W* who receives the invitation, choose the best boy among the inviters and become his "temporary" dance partner, and for all other inviters, refuse the invitation. We change the status of *W* to be "not free". As long as there are "free" girls, the algorithm performs the following steps:

In the new round, for each boy who is unmatched in the previous round, he will select the highest ranked girl who has not been invited by him according to his preference order and send her an invitation, regardless of whether this girl currently has a "temporary" dance partner. On the girl's side, if some girl has "temporary" dance partner, she pretends to receive the invitation from her partner in this round. Then, for each girl who receives at least one invitation in this round, choose the best boy among the inviters and become his "temporary" dance partner, and for all other inviters, refuse the invitation. We change the status of this girl to be "not free". The algorithm re-examines whether there are "free" girls, and start a new round if there are any "free" girls.

We give an example with 5 boys and 5 girls to illustrate how Gale-Shapley algorithm works. In the example shown in Fig. 4.13, boys are represented by numbers, and girls are represented by capital letters. On the left side of the figure, the letter string next to the number indicates the ranking of the boy's preference for all girls. On the right side of the figure, a string of numbers next to the letter indicates the ranking of girl's preferences. The rankings on both sides are arranged from most like to least like. For instance, the entry 1:CBEAD indicates that Boy 1 likes C the most and D the least.

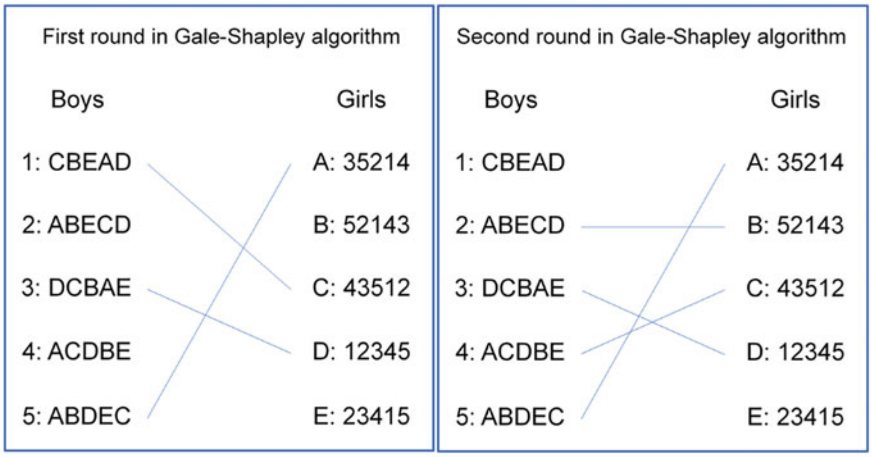


Boys	Girls
1: CBEAD	A: 35214
2: ABECD	B: 52143
3: DCBAE	C: 43512
4: ACDBE	D: 12345
5: ABDEC	E: 23415

Fig. 4.13 Example of the Gale-Shapley algorithm

In the first round, each boy will propose to the girl he likes most. Girl C will receive invitation from boy 1, and she becomes his "temporary" dance partner. Girl A will receive invitation from boy 2,4,5 and she will become the "temporary" dance partner of boy 5 according to her preference. Girl D will receive invitation from boy 3, and she becomes his "temporary" dance partner. In the end of this end, girl B and E are still free while boy 2 and 4 do not have dance partner.

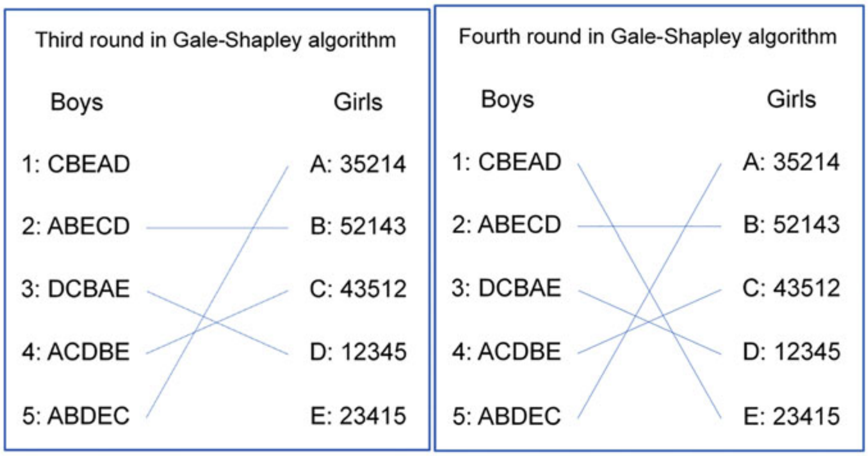
In the second round, boy 2 will propose to girl B and boy 4 will propose to girl C. For girl B, she only receives the invitation from boy 2, thus she becomes his "temporary" dance partner. But for girl C, she is currently the partner of boy 1 and receives a new invitation from boy 4. She will compare these two boys according to her preference, and becomes the partner of boy 4. In the end of this round, boy 1 becomes unmatched.



In the third round, boy 1 will propose to girl B. But girl B thinks her current partner boy 2 is better than boy 1, so she will refuse the invitation.

In the fourth round, boy 1 will propose to girl E. Since girl E is free, she will accept the invitation. Now, all girls become "not free", thus, the algorithm terminates.

The reader can verify that {(1, E), (2, B), (3, D), (4, C), (5, A)} is indeed a stable matching.



Now, let us firstly discuss the correctness of Gale-Shapley algorithm. It is not always obvious whether an algorithm correctly solves the problem we require, especially for some complex algorithms. However, it is quite important to strictly prove the correctness of any algorithm, otherwise, there is no guarantee for the output. The previous algorithms based on divide-and-conquer method are relatively simple, and the correctness of the algorithm is self-evident, so we omitted the proof of the correctness. But for the Gale-Shapley algorithm, the correctness is not obvious. It is even not obvious why the algorithm will eventually terminate. From a mathematical point of view, "stable matching must exist" is not a clearly established proposition.

Before proving the correctness of Gale-Shapley algorithm, we first observe some simple properties of this algorithm:

1. Every boy invites a girl at most once;
2. Every girl keeps the status "not free" since she was first invited;
3. Every girl has at most one dance partner during the process of the algorithm;
4. Every boy has at most one dance partner during the process of the algorithm;
5. Every unmatched boy will continue to invite until it matches or he has invited all girls;

The following lemma shows the correctness of the Gale-Shapley algorithm

Lemma 1: Gale-Shapley algorithm stops after $O(n^2)$ rounds, and after it stops, it will output a matching.

Proof: According to property 1, we know that each boy invites n times at most, so the total number of invitations is at most n^2. In each round, there are at least one invitation. Thus, the algorithm will terminate after at most n^2 rounds

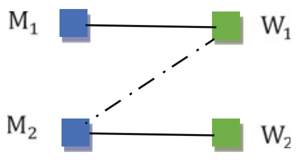
When the algorithm stops, if all girls are "not free", according to property 3 and 4, all girls and boys are matched. Thus it forms a matching.

It seems that we have already finished the proof. However, there are some subtlety in the algorithm. In the algorithm, in each round, we say "for each unmatched boy, he will select the highest ranked girl who has not been invited by him and send the invitation". But, does it possible that for some unmatched boy, he has already invited all girls? We will show this case is impossible.

We show it by contradiction. Suppose for some boy M in the beginning of some round T, he is unmatched, but he has already invited all girls. According to property 2, after a girl receives her first invitation, her status will always be "not free". Since this boy M has invited all girls, the status of all girls should be "not free" in the beginning of round T. Thus, the algorithm should stop in the previous round. Contradiction. We finish the proof. ∎

Lemma 2: The output by the Gale-Shapley algorithm is a stable matching.

Proof: We will still prove it by contradiction. Suppose the final matching output by the Gale-Shapley algorithm is $\{(\mathrm{W_1, M_1}), (\mathrm{W_2, M_2}), \ldots, (\mathrm{W_n, M_n})\}$. Without loss of generality, let us assume that the unstable pair in the final matching is (W_1, M_2). This pair is unstable means W_1 prefers M_2 to M_1, and M_2 prefers W_1 to W_2 (see figure below).



We consider two cases:

Case 1: M_2 has never invited W_1. Since M_2 and W_2 are finally together, it means that M_2 invites W_2 in some round. On the other hand, since M_2 prefers W_1 to W_2, M_2 must invite W_1 before he invites W_2. Contradiction.

Case 2: M_2 has invited W_1 in some round. Since M_1 and W_1 are finally together and M_2 has invited W_1 in some round, it means W_1 refuses the invitation of M_2 in some round due to she receives invitation from some better boy. For every girl, she will refuse the invitation or change partner only if some better boy sends her invitation, so it means in the girls' view, their partners becomes better and better. Since the final partner of W_1 is M_1, girl W_1 prefers M_1 to M_2. Contradiction.

Therefore, in each case, we will reach contradiction. We finish the proof. ∎

Question: in your opinion, is this algorithm beneficial for boys or girls?

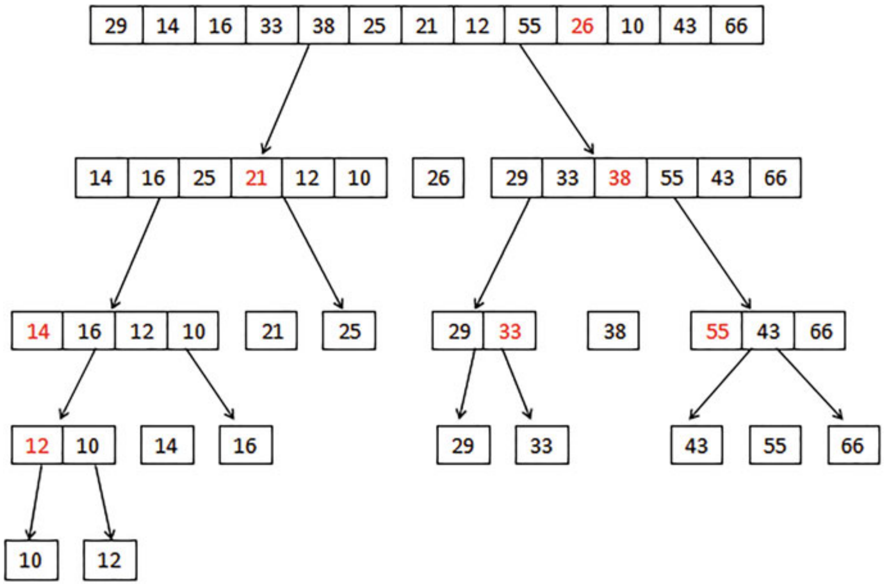


Fig. 4.14 An example run of the quicksort algorithm

4.3.3 *The Randomization Strategy*

In the previous section, we have already introduced the sorting problem and discussed three sorting algorithms: the bubble sort algorithm, the insertion sort algorithm and the merge sort algorithm. In addition to these sorting algorithms, there are many different sorting algorithms. In this section, we will introduce a sorting algorithm commonly used in our computers: the **quicksort** algorithm. The difference is that we will use randomized process in the algorithm, and we will show the power of randomization.

The core idea of the quicksort algorithm is similar to the merge sort algorithm: call itself recursively to sort the subproblems. In the merge sort algorithm, we firstly solve the subproblems and then try to merge the results into a whole ordered set. But in the quicksort algorithm, we will carefully divide the original problem into subproblems, and after solving the subproblem, the merge process becomes trivial. How can we achieve this? Suppose the original array is A. The key idea is to divide A into two subsets A_1 and A_2 where all elements in A_1 are smaller than all elements in A_2, then the merge process will be trivial.

```
QuickSort(A, p, r)
  If $p < r$
    1. $q$ = Partition(A, $p$, $r$)
    2. QuickSort(A, $p$, $q$-1)
    3. QuickSort(A, $q$+1, $r$)
```

The above is the pseudocode of the quicksort algorithm, which sorts the p-th element to the r-th element in the array A. It needs to call a Partition subroutine.

The Partition (A, p, r) subroutine uniformly and randomly extracts an element *x* from the array A[p,. . ., r], and then adjusts the array A[p,. . . r] so that the numbers larger than *x* are arranged on the right side of *x*, and the numbers smaller than *x* are arranged on the left side of *x*. Note that the numbers on the right side are not sorted, and the same goes for the left side. Partition(A, p, r) finally returns the position *q* of *x* in the array.

After the operation of the Partition() subroutine, we know that any number on the left side of *x* must be smaller than the one on the right side, so we only need to sort the elements A[p,. . ., q-1] on the left side of *x* and the elements A[q + 1, . . ., r] on the right side of *x*, separately. We can do so by recursively call QuickSort() for the two subproblems.

Figure 4.14 shows an example of the specific implementation of the quicksort algorithm. The elements which are randomly selected each time are marked in red in the figure.

Part of the Go code to implement the above quicksort algorithm is shown below, for students who want more details. Note that initially, the input data is stored in a slice variable A.

```
 func quicksort(A []int) {
   if len(A) < 2 { return }
   lowerA, upperA := partition(A)
   quicksort(lowerA)
   quicksort(upperA)
 }
 func partition(A []int) ([]int, []int) {      // return two slices as
output
   pivotIndex := rand.Intn(len(A))      // randomly select a pivot
   pivotValue := A[pivotIndex]
   lower := 0
   A[pivotIndex], A[len(A)-1] = A[len(A)-1], A[pivotIndex]
   for i:= 0; i<len(A); i++ {
     if (A[i]<pivotValue) {
       A[lower], A[i] = A[i], A[lower]
       lower++
     }
   }
   A[lower], A[len(A)-1] = A[len(A)-1], A[lower]
```

```
    return A[0:lower], A[lower+1:len(A)]
  }
```

Finally, let us analyze the performance of the quicksort algorithm. Because the Partition() subroutine called in the quicksort algorithm selects the elements randomly, the running time of the quicksort algorithm is not deterministic but is a random variable. We use $T(n)$ to represent the time required by the quicksort algorithm to sort n unordered numbers, thus $T(n)$ is a random variable.

At best, if each time the Partition() subroutine happens to divide the array into two equal parts, the total sorting time will be very fast. We use $T^{best}(n)$ to represent the time of the algorithm in this lucky case. Then we have

$$T^{best}(n) = 2T^{best}\left(\frac{n}{2}\right) + n.$$

By solving the recursion, we have $T^{best}(n) = O(n \log n)$.

However, if the length of each part is very uneven, then the algorithm will be very slow. For example, in the extreme cases, the algorithm will always choose the largest elements each time. In this case, one of the parts after the partition is the empty set, while the other part contains $n - 1$ element. We use $T^{worst}(n)$ to represent the time of the algorithm in this unlucky case. Then we have

$$T^{worst}(n) = T^{worst}(n - 1) + n.$$

By solving the recursion, we have $T^{worst}(n) = O(n^2)$.

So which one is better to represent the performance of the quicksort algorithm? Usually, we use neither **best case** analysis nor **worst case** analysis. Instead, our goal is to analyze the **average case** running time, i.e., the expectation of $T(n)$.

Since we select an element uniformly and randomly every time, the probability of selecting any element is $1/n$. Assuming that the element x selected by the algorithm is ranked i among all the elements of the array, then there are $(i - 1)$ elements smaller than x where these elements will be ranked on the left of x and the time required to recursively call the QuickSort() algorithm is $T(i - 1)$. Similarly, $(n - i)$ elements are larger than x and will be sorted to the right of x, and the time required to recursively call QuickSort() algorithm is $T(n - i)$. Therefore, the expectation of total sorting time $T(n)$ is

$$\mathbb{E}(\mathrm{T}(n)) = \frac{1}{n}\sum\nolimits_{i=1}^{n}\mathbb{E}(\mathrm{T}(i - 1) + \mathrm{T}(n - i) + n - 1).$$

The last item $(n - 1)$ is due to the need to compare x with all other elements. After simplifying the above formula, we can get

Table 4.5 The dictionary StudentsMap for linear search

Key	Value
Berners-Lee, Tim	UK
Wu, Wenjun	China
Godel, Kurt	Austria
Turing, Alan	UK
Knuth, Donald	USA
Leibniz, Gottfried	Germany
Von Neumann, John	Hungary
Amdahl, Gene	USA
Yao, Andrew	China
Moore, Gordon	USA
Yang, Xiong	China
Karp, Richard	USA
Boole, George	UK
Makimoto, Tsugio	Japan
Torvalds, Linus	Finland

$$\mathbb{E}(\mathrm{T}(n)) \leq \frac{2}{n}\sum_{i=1}^{n-1}\mathbb{E}(\mathrm{T}(i)) + (n-1).$$

By solving the recursion, we have $\mathbb{E}(\mathrm{T}(n)) = O(n\log n)$. In other words, the expected running time of the quicksort algorithm is $O(n \log n)$, which is significantly faster than the $O(n^2)$ time required by the bubble sort algorithm and insertion sort algorithm in the previous section.

4.3.4 (***) Search Algorithms

When computer science is used to solve a real-world problem, it often happens that the given problem has different nuances, which can be utilized to design different algorithms. As a concrete example, we discuss the search problem by contrasting three algorithms to look up an item in a dictionary. The three algorithms have different time complexities of O(n), O(log$n$), and O(1).

A **dictionary** is a set of n records, where each record consists of a key-value pair <*key*, *value*>. A **search** operation lookup(*key*) returns the *value* of the key-value pair with a matching *key*.

A dictionary of students of computer science, who are actually pioneers of computer science quoted in the book, is shown in Table 4.5. There are n=15 records, each denoting a pioneer. The key is the full name of a student, and the value is the country of birth of the same student, both of which are strings.

Table 4.6 Initial configuration of the search space for binary search

	Index	Key	Value
low	0	Amdahl, Gene	USA
	1	Berners-Lee, Tim	UK
	2	Boole, George	UK
	3	Godel, Kurt	Austria
	4	Karp, Richard	USA
	5	Knuth, Donald	USA
	6	Leibniz, Gottfried	Germany
mid →	7	Makimoto, Tsugio	Japan
	8	Moore, Gordon	USA
	9	Torvalds, Linus	Finland
	10	Turing, Alan	UK
	11	Wu, Wenjun	China
	12	Von Neumann, John	Hungary
	13	Yang, Xiong	China
high	14	Yao, Andrew	China

4.3.4.1 Linear Search in O(n) Time

The first algorithm is **linear search**, which searches the dictionary record-by-record and is implemented by the following linear.search.go program. A data structure called *map* is used to represent a dictionary, where the keyword *range* says that the for loop iterates record-by-record for studentsMap. Thus, lookup("Knuth, Donald") needs 5 iterations and returns "USA", and lookup("Babayan, Boris") needs 15 iterations and returns "not found".

```
package main   // The linear.search.go program
import "fmt"
var studentsMap = map[string]string{
    "Berners-Lee, Tim": "UK", "Wu, Wenjun": "China",
 "Godel, Kurt": "Austria", "Turing, Alan": "UK",
  "Knuth, Donald": "USA", "Leibniz, Gottfried": "Germany",
  "Von Neumann, John": "Hungary", "Amdahl, Gene": "USA",
  "Yao, Andrew": "China", "Moore, Gordon": "USA",
  "Yang, Xiong": "China", "Karp, Richard": "USA",
  "Boole, George": "UK", "Makimoto, Tsugio": "Japan",
  "Torvalds, Linus": "Finland",
}
func lookup(studentName string) string {
  for key, value := range studentsMap {
    if studentName == key { return value }
  }
  return "not found"
}
func main() {
  s, t := "Knuth, Donald", "Babayan, Boris"
  fmt.Println(s, "is from", lookup(s))
```

```
    fmt.Println(t, "is", lookup(t))
  }
```

4.3.4.2 Binary Search in O(log*n*) Time

A more efficient algorithm is **binary search**, which applies to a pre-sorted dictionary, as shown in Table 4.6. The main idea is to narrow down the search space by half with each iteration. Looking up a record with an input key *K* in a sorted *n*-element array *A* needs only O(log*n*) iterations. The algorithm goes as follows:

```
Initially: low, high = 0, n-1        // indices = [0, n-1]
while indices not empty
  mid = (low+high)/2
  if K == A[mid].key then return A[mid].value
  if K < A[mid].key then high=mid-1   // indices = [low, mid-1]
  else low=mid+1          // indices = [mid+1, high]
return "not found"
```

Binary search has the following notable differences from linear search.

- The dictionary is stored in an array with explicit index. The array is presorted by Key. That is, if j>i, then A[j].key > A[i].key. For instance, 14>8 and A[14].key > A[8].key, because A[14].key = "Yao, Andrew", A[8].key="Moore, Gordon".
- In each iteration, the input *K* is compared to the *Key* of the middle element. If there is a match, the binary search algorithm returns the *Value* of the middle element and stops. If there is a mismatch, the algorithm cuts the search space by half (by adjusting low or high) and goes to the next iteration.
- If no value is returned after all iterations, the input key *K* does not match the key of any of the array elements. The algorithm outputs "not found".

For instance, to look up "Knuth, Donald", the search space evolves as follows. At the first iteration, "Knuth, Donald"<"Makimoto, Tsugio", update *high* to 6.

In Iteration 2: "Knuth, Donald">"Godel, Kurt", update *low* to 4

	Index	Key	Value
low	0	Amdahl, Gene	USA
	1	Berners-Lee, Tim	UK
	2	Boole, George	UK
mid →	3	Godel, Kurt	Austria
	4	Karp, Richard	USA
	5	Knuth, Donald	USA
high	6	Leibniz, Gottfried	Germany

In Iteration 3: "Knuth, Donald"="Knuth, Donald", found; return "USA"

	Index	Key	Value
low	4	Karp, Richard	USA
mid →	5	Knuth, Donald	USA
high	6	Leibniz, Gottfried	Germany

It takes 3 iterations to look up "Knuth, Donald" and outputs "USA". What if the input key K does not match any key of the array? Then log*n* iterations are needed. For instance, it takes log(15) = 4 iterations to look up "Babayan, Boris" and outputs "not found".

When implementing the binary search algorithm in a Go program, we need to pay special attention to making sure that the given dictionary is a presorted array. That is, the Key field is ordered. We use a *struct* type to define an array element.

```
const n = 15
var studentsArray = [n] struct {
  key    string    //studentName
  value  string    //studentCountry
}
```

Variables key and value are ASCII strings. "Gödel, Kurt" must be written as "Godel, Kurt", since ö is not an ASCII character. "von Neumann, John" must be written as "Von Neumann, John", since ‘v’ has a large ASCII encoding than capitals.

The complete binary.search.go program follows.

```
package main
import "fmt"
const n = 15
var studentsArray = [n] struct {
    key    string
    value  string
}{
  {"Amdahl, Gene", "USA"}, {"Berners-Lee, Tim", "UK"},
  {"Boole, George", "UK"}, {"Godel, Kurt", "Austria"},
  {"Karp, Richard", "USA"}, {"Knuth, Donald", "USA"},
  {"Leibniz, Gottfried", "Germany"}, {"Makimoto, Tsugio", "Japan"},
  {"Moore, Gordon", "USA"}, {"Torvalds, Linus", "Finland"},
  {"Turing, Alan", "UK"}, {"Von Neumann, John", "Hungary"},
  {"Wu, Wenjun", "China"}, {"Yang, Xiong", "China"},
  {"Yao, Andrew", "China"},
}
func lookup(studentName string) string {
  var low, high, mid int
  low, high = 0, n-1
  for low <= high {
      mid = ( low + high ) / 2
      if studentName == studentsArray[mid].key {
       return studentsArray[mid].value
      }
     if studentName > studentsArray[mid].key {
       low = mid + 1
      } else { high = mid - 1 }
  }
  return "not found"
}
func main() {
  s, t := "Knuth, Donald", "Babayan, Boris"
  fmt.Println(s, "is from", lookup(s))
  fmt.Println(t, "is", lookup(t))
}
```

The screen outputs are:

```
> go run ./binary.search.go
Knuth, Donald is from USA
Babayan, Boris is not found
>
```

4.3.4.3 Hash Search in O(1) Time

Binary search (O(log$n$)) is much faster than linear search (O(n)). However, some application scenarios need an even faster algorithm with constant complexity, i.e., O (1). For instance, when one registers for an Internet service, the system may need to instantly check against a trillion-record dictionary, to see if a particular user name, e.g., "johnSmith", is already chosen by another user. When one compiles a document, the document-writing software system constantly checks against a million-record English dictionary, to see if a word just entered is misspelled.

A method called **hashing** can help achieve this goal. The hash search algorithm is based on three basic observations.

- First, although the number of keys and records in the dictionary may be quite large, the number of keys actually stored is much smaller.
- The keys actually stored can be organized as a **hash table**, such that a **hash index** can be computed in O(1) time from an input key by a **hash function**, to directly access an element of the hash table. That is, a lookup operation needs only O (1) time, when we are lucky.
- When we are not lucky, several keys may map to the same hash index, a situation called *collision*. The collided keys need to be further organized, e.g., as a linked list. If most of search time is spent on looking through a linked list, the *worst-case* time complexity for lookup becomes O(n). However, computer science has produced optimized hash search algorithms, such that the *average* time complexity for lookup becomes O(1).

Here we use the concept of **average time complexity**. A dictionary, once produced, will often be looked up many times. Suppose there are m lookup operations in total. Some lookup operations take O(n) time, and some take only O(1) time. Assume the i-th lookup operation takes $T(i)$, where $1 \leq i \leq m$. Then the average time complexity for a lookup operation is $\left(\sum_{i=1}^{m} T(i)\right)/m$. Suppose 1 trillion lookup operations are performed. One million of them each take O(n) time, and the rest each take O(1) time. Then, for each lookup operation, the worst-case time complexity is O(n), but the average time complexity becomes only O(1).

As a concrete example of hashing, suppose a legitimate user name consists of 10 digits and letters, such as "johnSmith9". Then the number of possible user names is huge ($62^{10} \approx 8.4 \times 10^{17}$). However, the number of user name strings actually stored could be much smaller, say 1 million. The actually stored keys (user names) and records are organized as a hash table of 1 million elements.

For the studentsMap example, a key (e.g., "Knuth, Donald") is the name of a computer science student, family name first. Again, the possible number of keys is huge. Assume the number of keys and records actually stored is 15 and the hash table has 15 elements. Then there will be no collision. Assume a more realistic case where the number of keys and records actually stored is 15, and the hash table has only 6 elements. Then there will be collisions, and each group of collided records is organized as a linked list.

Similar to linear search, the hash search algorithm is given as input a dictionary of key-value pairs shown in Table 4.5, and a Key to look up. The output is the pairing Value, when the Key is found in the dictionary, or "not found" if otherwise. To implement the hash search algorithm in a Go program hash.search.go, we need to pay attention to the following details.

- Implement a linked list as a number of records connected by pointers.
- Implement a hash table as an array of such records.
- The value of an array index is computed by a hash function.
- The main function first fills up the hash table with record items.
- To lookup a Key such as "Knuth, Donald", first use its hash function output as index to access the array element of the hash table. If "Knuth, Donald" is found, return his country "USA". If not found there, continue traversing the linked list.

  The key-value pairs of Table 4.5 are initialized in a map variable studentsMap.

```
var studentsMap = map[string]string{
  "Berners-Lee, Tim": "UK", "Wu, Wenjun": "China",
  "Godel, Kurt": "Austria", "Turing, Alan": "UK",
  "Knuth, Donald": "USA", "Leibniz, Gottfried": "Germany",
  "Von Neumann, John": "Hungary", "Amdahl, Gene": "USA",
  "Yao, Andrew": "China", "Moore, Gordon": "USA",
  "Yang, Xiong": "China", "Karp, Richard": "USA",
  "Boole, George": "UK", "Makimoto, Tsugio": "Japan",
  "Torvalds, Linus": "Finland",
}
```

- (1) Implement a linked list as a number of records connected by pointers.

  The variable Record in hash.search.go is similar to studentsArray of the binary search example. They both use a struct data type to represent key-value pairs.

```
var studentsArray = [n] struct {
  key     string    // studentName
  value   string
}
```

However, variable Record is a three-field structure. In addition to the key-value pair, it has a *next* field, which is a pointer to the next Record in the linked list, where *Record denotes the memory address of a Record.

```
type Record struct {
  next            *Record
  studentName     string
  studentCountry  string
}
```

- (2) Implement a hash table as an array of such records.
  Variable hashTable denotes an array of 6 elements, with array indices 0, 1, 2, 3, 4, 5. Each element hashTable[i] holds a pointer to a Record.

```
const HashTableSize = 6
var hashTable [HashTableSize] *Record
```

- (3) The value of an array index is computed by a hash function.
  The element of an array A at index i is accessed by A[i]. In a hash table, the array element for studentName is accessed by first computing a hashFunction.

```
    hashIndex := hashFunction(studentName)
    entry := hashTable[hashIndex]
```

The hash function first finds the sum of ASCII values of characters in studentName, using code from the student name coding exercise. Then the modulus operator % is used to get the remainder of sum mod 6, where 6 is the size of the hash table.

```
func hashFunction(name string) int {
  sum := 0
  for i := 0; i < len(name); i++ { sum = sum + int(name[i]) }
  return sum % HashTableSize
}
```

For instance, given key="Berners-Lee, Tim", we have

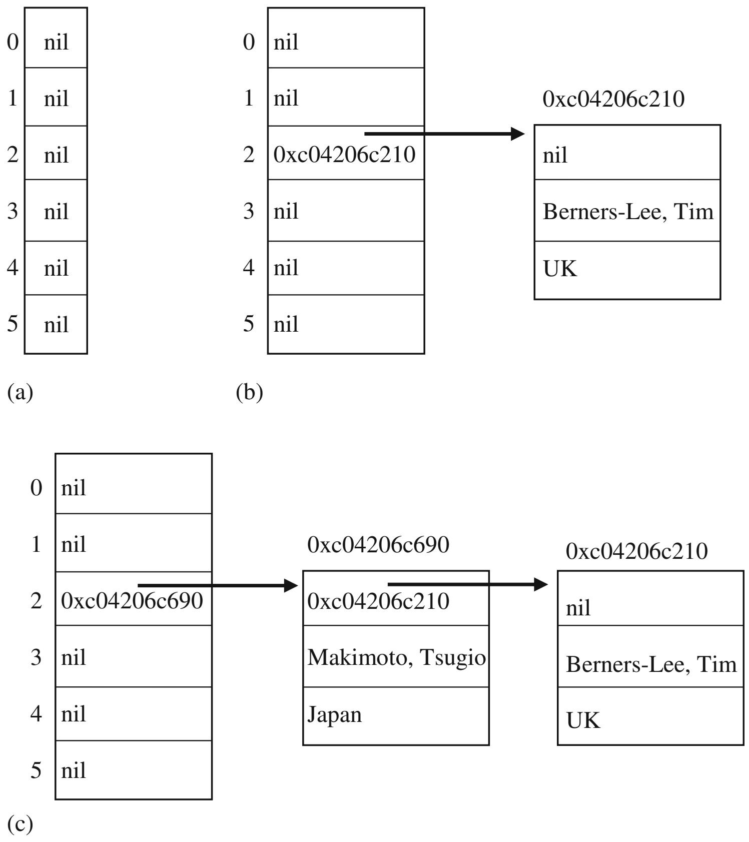


Fig. 4.15 The hash table with associated linked list for hashTable[2]. (**a**) Initial table. (**b**) After inserting record for Berners-Lee. (**c**) After inserting record for Makimoto

```
sum of "Berners-Lee, Tim" = 1418
sum % hashTableSize = 1418 % 6 = 2
```

Thus, hashFunction("Berners-Lee, Tim") returns 2.

- (4) Before any looking up, the hash table needs to be filled with record items.

  The action of filling out hashTable is done by the following code. Recall that the for loop ranges over studentsMap, one record (key-value pair) at a time. For each key-value pair, a new record is created and inserted to a hashTable element.

```
for key, value := range studentsMap {          // fill out hashTable
   hashIndex := hashFunction(key)
   newRecord := &Record{   // & denotes the address of Record
     next:          hashTable[hashIndex],
     studentName:      key,
     studentCountry:    value,
    }
    hashTable[hashIndex] = newRecord
  }
```

Initially, all elements of the array variable hashTable are initialized to the zero value of pointer, which is *nil*. That is, hashTable[i] contains value nil for all i.

Consider the case when the for loop goes to the key-value pair "Berners-Lee, Tim": "UK", where key= "Berners-Lee, Tim" and value= "UK". We know that the statement

```
hashIndex := hashFunction(key)
```

assigns 2 to hashIndex. The next few statements creates and inserts a new record for "Berners-Lee, Tim" to the element hashTable[hashIndex], that is, hashTable[2]. More specifically, we see the following execution steps, illustrated in Fig. 4.15.

Figure 4.15a shows the initial configuration of hashTable, where all elements contains nil: the pointer points to nowhere. Figure 4.15b shows the configuration after the record for Berners-Lee is created and inserted. This new record has an address newRecord, and its three fields have the following values:

```
newRecord.next:              nil
newRecord.studentName:       "Berners-Lee, Tim"
newRecord.studentCountry:    "UK"
```

The system allocates memory space for this record, which happens to assign address 0xc04206c210 to newRecord.

Figure 4.15c shows the configuration after the record for Makimoto is created and inserted. Note that this record is inserted at the front of the linked list. This new record has an address newRecord, and its three fields have the following values:

```
newRecord.next:              0xc04206c210
newRecord.studentName:       "Makimoto, Tsugio"
newRecord.studentCountry:    "Japan"
```

The address of this record, newRecord, is 0xc04206c690. This is an address automatically generated by the Go programming language system. It may change when the same program executes again. That is why we use a more abstract arrowed line to denote a pointer. Figure 4.16 shows the filled-out hash table and linked lists.

- (5) A lookup operation first finds the hash table array element, and then traverses the linked list.

  To look up the record for the input key "Knuth, Donald", we have

```
sum of "Knuth, Donald" = 1192; sum % hashTableSize = 1192 % 6 = 4
```

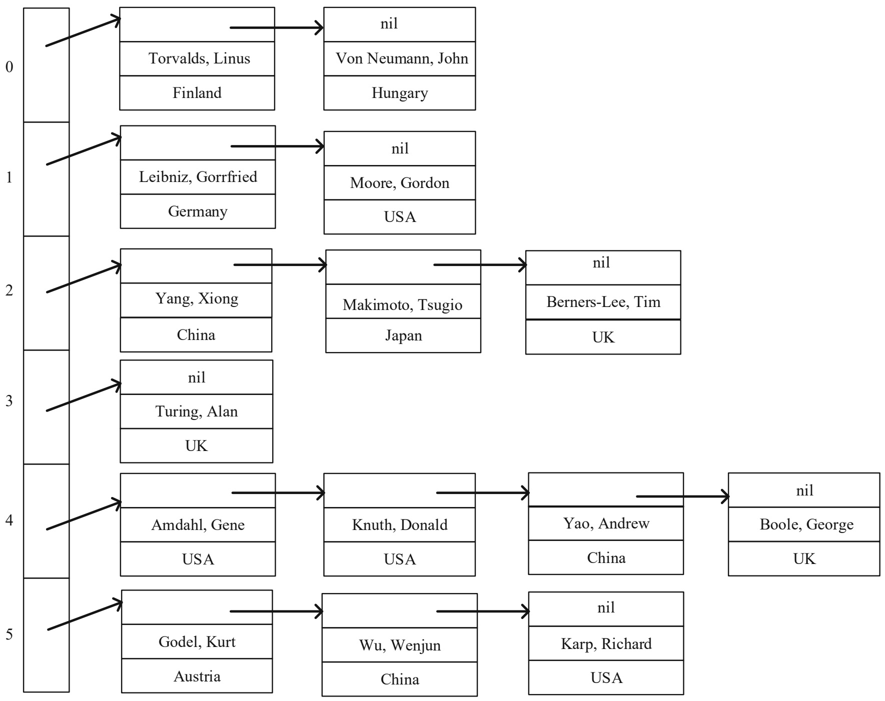


Fig. 4.16 The hash table with associated linked lists for studentsMap

Thus, the hash table element is hashTable[4], which points to the record for Amdahl. Since the key "Amdahl, Gene" does not match the input key "Knuth, Donald", the program goes to the next record by following the *next* field. Then we have a match.

The complete hash.search.go program follows.

```
package main
import "fmt"

type Record struct {
  next            *Record
  studentName     string
  studentCountry  string
}

const HashTableSize = 6
var hashTable [HashTableSize] *Record

func hashFunction(name string) int {
  sum := 0
  for i := 0; i < len(name); i++ { sum = sum + int(name[i]) }
  return sum % HashTableSize
}

func lookup(studentName string) string {
  hashIndex := hashFunction(studentName)
  entry := hashTable[hashIndex]
  current := entry
  for current != nil {
    if current.studentName == studentName {
      return current.studentCountry      // keys match
    }
    current = current.next             // otherwise goto next
  }
  return "not found"
}

var studentsMap = map[string]string{
  "Berners-Lee, Tim": "UK", "Wu, Wenjun": "China",
  "Godel, Kurt": "Austria", "Turing, Alan": "UK",
  "Knuth, Donald": "USA", "Leibniz, Gottfried": "Germany",
  "Von Neumann, John": "Hungary", "Amdahl, Gene": "USA",
  "Yao, Andrew": "China", "Moore, Gordon": "USA",
  "Yang, Xiong": "China", "Karp, Richard": "USA",
  "Boole, George": "UK", "Makimoto, Tsugio": "Japan",
  "Torvalds, Linus": "Finland",
}

func main() {
  for key, value := range studentsMap {          // fill out hashTable
    hashIndex := hashFunction(key)
    newRecord := &Record{
      next:          hashTable[hashIndex],
      studentName:      key,
      studentCountry:    value,
    }
    hashTable[hashIndex] = newRecord
  }
  s, t := "Knuth, Donald", "Babayan, Boris"       // look up s and t
  fmt.Println(s, "is from", lookup(s))
  fmt.Println(t, "is", lookup(t))
}
```

The screen outputs are:

```
> go run ./hash.search.go
Knuth, Donald is from USA
Babayan, Boris is not found
>
```

4.4 P vs. NP

In the previous sections, we have learned a variety of effective methods to design smart algorithm for many problems. We always try to solve a problem as fast as possible. For example, in the sort algorithm, while the bubble sort algorithm and the insertion sort algorithm need $O(n^2)$ time, the merge sort algorithm needs only $O(n \log n)$ time. We call $O(n^2)$ or $O(n \log n)$ the time complexity of the corresponding algorithm. Thus, the merge sort algorithm is more efficient.

One may wonder if it is possible to design an even faster algorithm for the sorting probm. In this section, we will briefly introduce *complexity theory* which focuses on the complexity of a computational problem, instead of any particular algorithm. We want to answer the following question systematically:

Are some computational problems inherently difficult to solve effectively?

In Sect. 4.4.1, we use the sorting problem as an example to illustrate the time complexity of a computational problem. We then introduce the most important complexity classes P and NP in Sect. 4.4.2. In the final section, we show several examples of P and NP.

4.4.1 Time Complexity

Intuitively, the time complexity of a computational problem is the minimum time complexity of any algorithm which can solve this problem. It is an important measurement to understand how difficult a problem is.

Consider the sorting problem discussed before. Since we have merge sort algorithm to solve it in $O(n \log n)$ time, the time complexity of sorting problem is at most $O(n \log n)$. But is there any other algorithm which can solve the sorting problem in $o(n \log n)$ time? This question is hard to answer since the answer depends on the usage scenario. We need a more precise model to discuss the hardness of the sorting problem.

Let us assume that the elements to be sorted can only be compared in pairs. That is, we can treat every element as a black box, the only way to know the order of element a and b is to compare them with one comparison step. Of course, we assume any pair of elements can be compared and the results form a total order over all elements.

Under these assumptions, we will show that any algorithm needs at least $n \log n$ steps of comparison operations in the worst case. The idea comes from the information theory. The number of all possible orders is $n!$, which is actually the number of all possible permutations over $\{1, 2, \ldots, n\}$. For each comparison step, the result will give us 1 bit information, either $a < b$ or $a > b$. Thus, in the worst case, if some algorithm can use k comparison steps to distinguish all possible orders, it means that $2^k \geq n!$. Thus, the time complexity of any algorithm which can solve sorting problem is $\Omega(n \log n)$. We call it the lower bound of time complexity for sorting problem, and it illustrates how hard the problem is.

Thus, the time complexity for the sorting problem is $\Theta(n \log n)$. We say the merge sort algorithm is asymptotic optimal.

4.4.2 P and NP

The important work of computational complexity theory is to determine the time complexity and space complexity of various problems. But it is difficult to decide the exact time complexity for all problems, so the researchers create many complexity classes where each complexity class contains various problems whose complexity are similar in some ways. P and NP are two most famous complexity classes in the computational complexity theory.

Intuitively, the complexity class P contains all decision problems which have polynomial time algorithms. Here, decision problem means that the output of the problem is either YES or NO, and the polynomial time algorithm means the time complexity is a polynomial function over the size of input.

Complexity class P contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time.

In practical applications, people usually refer to polynomial time algorithms as effective algorithms. Therefore, an important task in the field of computational complexity is to determine whether certain computational problems have polynomial time algorithms. In other words, decide whether such problems belong to P or not. For example, the sorting problem is in P since it has a polynomial time algorithm. The bubble sort algorithm, insertion sort algorithm, merge sort algorithm and quicksort algorithm are all polynomial time algorithms.

However, people gradually discovered that many basic problems are in such a gray area: we have neither found polynomial time algorithms to solve these problems, nor have we been able to prove that such algorithms do not exist. These problems involve a wide range of areas, distributed in various fields: operational research, optimization, combinatorics, logic, artificial intelligence, big data processing, and so on. Even after more than half a century of development, theoretical computer scientists are still at a loss for this gray area, and most of their problems still stay in this gray area. However, researchers have made great progress in the characterization of these problems and found that a large class of these problems belongs to NP, another important complexity class we introduce next.

Complexity class NP contains all decision problems that can be solved by a *non-deterministic* Turing machine using a polynomial amount of computation time.

However, this definition is slightly difficult to use. We will not introduce non-deterministic Turing machine in this book. The interested readers please refer to any complexity theory textbook. Here, we introduce an equivalent definition of the class NP.

Complexity class NP (equivalent definition): contains all decision problems that whose "YES" answers can be verified in polynomial time by a *deterministic* Turing machine. That is, there exists a checking algorithm which can verify the correctness of "YES" answer with the help of a witness in polynomial time.

More precisely, a decision problem $A \in$ NP, if and only if there exists a polynomial time algorithm S which is a checking algorithm and two constants c, c' such that the following two conditions hold:

1. For any input instance x whose correct answer should be "YES", there exists a witness y such that $|y| \leq c'|x|^c$ and if the algorithm S takes (x, y) as the input, it will output "YES";
2. For any input instance x whose correct answer should be "NO", for any witness y who satisfies $|y| \leq c'|x|^c$, if the algorithm S takes (x, y) as the input, it will always output "NO".

Now, let us use a concrete example to help understand the definition of NP.

Example 4.6. The Subset Sum Problem

Consider the following problem: given $2n$ integers, we want know whether it can be partitioned into two groups whose sums are the same. The integers are not required to be different from each other.

For example, if the input is {1,2,3,4,5,5}, the answer is YES since partition {1,2,3,4} and {5,5} have the same sum. If the input is {1,2,3,4,5,6}, the answer is NO since the sum of all integers is 21 which is an odd number.

This problem is quite important in cryptograph. However, till now, we do not know how to solve this problem in polynomial time over n. The naïve algorithm is to check all possible partitions. The time complexity of the naïve algorithm is $O(n2^n)$ which is exponential over n. On the other hand, we do not know how to prove that such problem cannot be solved in polynomial time. This is one example of the problems in the gray area we mentioned before. Here, we want to show that the subset sum problem is in NP.

To show that some problem belongs to NP, we need to construct the checking algorithm S and for any YES instance, construct the witness. For the subset sum problem, if x is a YES instance, we construct the witness $y = (y_1, y_2)$ where (y_1, y_2) is a partition of input x and the sums of y_1 and y_2 are the same. The checking algorithm S is designed to check two things: 1) the witness $y = (y_1, y_2)$ is a partition of input instance x, i.e., every element in x appears exactly once in either y_1 or y_2; and 2) the sum of integers in y_1 is the same as the sum of integers in y_2. Obliviously, the size of witness is a polynomial over the size of input, and the checking algorithm also runs in polynomial time. It is also easy to show that for any NO instance, it is impossible to construct a witness which can pass the checking algorithm. Thus, we have shown that the subset sum problem is in NP.

For any decision problems in P, it is also in NP, since we can compute the correct answer for any instance within polynomial time and we do not need witness at all. Thus, we have P $\subseteq$ NP. In the computer science field, one of the most fundamental and far-reaching issues is whether the opposite direction holds or not, that is, if some decision problem can be verified efficiently, is it always be computed efficiently? The problem is called P versus NP problem:

P versus NP problem: is P=NP?

This question is one of the seven millennium prize problems. At present, among these seven problems, only the Poincaré conjecture has been solved by the mathematician Grigori Perelman in 2003. The remaining six problems have not been solved. Although it is still unconfirmed whether P is equal to NP, most scientists believe that the equality does not hold, that is, there exists some decision problem which can be efficiently verified but cannot be efficiently computed.

4.4.3 (***) Examples in the NP Class

As we mentioned before, in the complexity class NP, there are many fundamental problems which we do not know whether they belong to P or not. We discuss two more examples of them: the graph coloring problem and the Hamiltonian path problem.

Example 4.7. The Graph Coloring Problem
The graph coloring problem can be specified as follows:

- **Input**: a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, an integer k;
- **Output**: decide whether there is a way to color each vertex with one of the k colors so that no adjacent vertices are of the same color.

Similar to the subset sum problem, the computer science community still does not know how to solve this problem in polynomial time over $n = |V|$, $m = |E|$ and k. The naïve algorithm is to check all possible coloring methods and the time complexity is $O(mn^k)$, which makes it an exponential time algorithm.

We now show the graph coloring problem belongs to NP. The witness of a YES instance is the valid way of coloring $f\colon V \to \{1, 2, \ldots, k\}$. The size of the witness is definitely polynomial over the size of input. The checking algorithm is straightforward. We only need to check for each edge, two endpoints do not have the same color. It is easy to check if the graph cannot be colored within k colors, it is impossible to find a valid way of coloring. Thus, Graph coloring problem belongs to NP.

☷

Example 4.8. The Hamiltonian Path Problem
The Hamiltonian path problem can be specified as follows:

- **Input**: a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges;
- **Output**: decide whether there is a Hamiltonian path in this graph. A Hamiltonian path is a path which visits each vertex exactly one.

Again, we do not know how to solve this problem in polynomial time over $n = |V|$. The naïve algorithm is to check all possible paths so that the time complexity is $O(n \cdot n!)$ which is exponential over n.

We show that the Hamiltonian path problem belongs to NP. The witness of a YES instance is the valid Hamiltonian path V_1, V_2, . . ., V_n which can also be seen as a permutation over V. The size of the witness is definitely polynomial over the size of input. The checking algorithm is also straightforward. We need to check two things: firstly, there is an edge between V_i and V_{i+1} for $i = 1, 2, \ldots, n - 1$; secondly, $V_1, \ldots, V_n$ are different from each other. It is easy to check if the graph does not contain a Hamiltonian path, it is impossible to find some witness which can pass the checking algorithm. Thus, Hamiltonian path problem belongs to NP.

In these two examples, it seems trivial to show that their decision problems belong to the complexity class NP. Actually, they not only belong to the class NP, but also are the most difficult problems in NP. They are **NP-complete** problems. If you can find a polynomial time algorithm for either the graph coloring problem or the Hamiltonian path problem, then every decision problem in NP has a polynomial time algorithm, which means P=NP. On the other hand, if you can prove that either the graph coloring problem or the Hamiltonian path problem does not have a polynomial time algorithm, you show P≠NP. So, if you are interested in P versus NP problem, you do not need to consider a class of problem. Instead, you can just think about one particular decision problem, for example, the graph coloring problem or the Hamiltonian path problem. Any ideas?

4.5 Exercises

1. Refer to Example 4.1. Show that Euclid's algorithm indeed computes the result gcd(x, y)=12 in three steps, given inputs x=36 and y=24.
2. Refer to Example 4.1. Show that Euclid's algorithm is indeed an algorithm, because it satisfies Knuth's five properties.
3. An algorithm must have at least one output, but it may not have any input. Please give an example of a meaningful algorithm without any input.
4. Which of the following statement is correct?
   (a) $0.1n^2$ is O(n)
   (b) $10000n$ is O(n^2)
   (c) $n \log n$ is $\Omega(n)$
   (d) $10n^2 - 10n + 1$ is $\Theta(n)$
5. Which of the following statement is correct?

(a) $\frac{n^3}{\ln n} + n^2$ is $O(n^2)$
(b) $n^2 + \frac{2n^2 \log n}{\log \log n}$ is $\Theta(n^2 \log n)$
(c) 2^{n^2} is $\Omega(n)$
(d) $(\ln n)^{\ln n}$ is $O(n^{100})$

6. Please sort the following asymptotic formulas from small to large: $\Theta(\log n)$, $\Theta(n)$, $\Theta\left(\frac{n}{\log n}\right)$, $\Theta(n \log n)$.
7. Please sort the following asymptotic formulas from small to large: $\Theta((\log n)^n)$, $\Theta(n^{100})$, $\Theta(n^{\log n})$, $\Theta((\log n)!)$.
8. The Sunway TaihuLight supercomputer was the world's fastest supercomputer from June 2016 to June 2018. It can finish 93 million billion operations per second. If we use this supercomputer to compute Steiner tree problem with 1000 vertices, how long do we need? The best algorithm for Steiner tree problem runs in $n^{\log_2 n}$ time where n is the number of vertices.
9. Given an unsorted array [1, 3, 9, 7, 6, 7, 8, 5, 2, 4], please describe the executing processes of the following sorting algorithms: bubble sort, insertion sort, merge sort and quicksort algorithms.
10. The insertion sort algorithm and the merge sort algorithm have a time complexity of $O(n^2)$ and $O(n \log n)$, respectively. What is the main reason that merge sorting is more efficient than insertion sorting?
11. For the single-factor optimization problem, why is that the equal division algorithm is not as efficient as the golden section algorithm? The textbook stated that it is smart to divide the problem into two equal sized subproblems.
12. For the integer multiplication problem, what is the main reason that the naïve algorithm of divide and conquer is not efficient?
13. Refer to the integer multiplication problem in Sect. 4.2.4. If you divide two numbers into 3 segments (each segment is an n/3-digit number), is it possible to find a faster algorithm for integer multiplication?
14. Consider the sorting problem of n numbers. $\underline{\text{In the worst case}}$, how many comparisons do we need in the quicksort and the bubble sort algorithms?

(a) $O(n \log n)$, $O(n^2)$
(b) $O(n^2)$, $O(n^2)$
(c) $O(n)$, $O(n \log n)$
(d) $O(n \log n)$, $O(n \log n)$

15. Given a sorted array with n numbers, what is the running time if we want to check whether element x and y are in this array?

(a) $\Theta(1)$
(b) $\Theta(\log n)$
(c) $\Theta(n/\log n)$
(d) $\Theta(n)$

16. Solve the following recursion: $T(n) = (\ )$

$$\begin{cases} T(1) = 1 \\ T(n) = 2T(n-1) \end{cases}$$

17. Solve the following recursion: $T(n) = (\ )$

$$\begin{cases} T(1) = 1 \\ T(n) = 3T\left(\frac{n}{2}\right) + n^2 \end{cases}$$

18. Solve the following recursion: $T(n) = (\ )$

$$\begin{cases} T(1) = 1 \\ T(n) = 2T\left(\left[\frac{n}{\sqrt{2}}\right]\right) + n^2 \end{cases}$$

19. 128 students take part in a table tennis match. Assume the ability of the students forms a total order. If we want to decide the champion, how many matches do we need? If we want to decide the champion and runner-up, how many matches do we need?

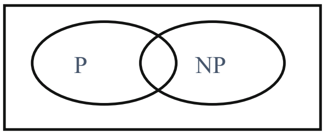
    (a) 127, 128
    (b) 127, 133
    (c) 127, 192
    (d) 127, 253

20. There are 16 bottles of liquid and one of them is poisonous. The poisonous one can make the mouse die immediately. Now we want to know which bottle is poisonous. Each time, we can mix the liquid from several bottles and let one mouse drink it. For each mouse, it can only drink once. In the worst case, how many mice do we need to find the poisonous bottle?
21. Suppose you are in a skyscraper and have one egg in your hand. You want to know from which floor can the egg fall without breaking. If you test some floor but the egg is broken, you have no egg to test anymore. Thus, the only feasible solution is to test one more floor each time. Now, consider you have two identical eggs. How can you achieve the goal with as fewer number of tests as possible? You can assume that the egg must be broken if it falls from the top floor which is the n-th floor.
22. In the stable matching problem, is the stable matching unique? If it is not unique, please give an instance where there are at least two different stable matching.
23. In the stable matching problem, is the Gale-Shapley algorithm beneficial for boys or girls? Why?
24. Does the following decision problem belong to NP?

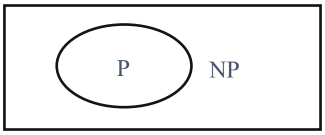
    - Problem: Given 2n integers, decide whether we can partition them into two sets (each set contains n integers) where the sum of two sets is equal.

25. Does the following decision problem belong to NP?
    - Problem: Given two integers x and y, decide if x is a multiple of y.
26. What is the relation between P and NP? Assume the rectangle represents all decision problems which can be computed by Turing machine.

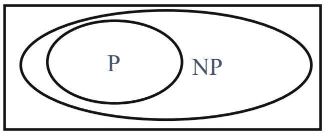
    (a)



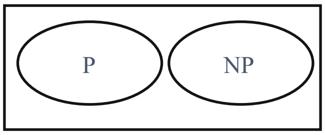
    (b)



    (c)



    (d)



4.6 Bibliographic Notes

The chapter quotation is from Professor Brian Kernighan of Princeton University [1]. Professor Donald Knuth of Stanford University gave a five-point definition of algorithm [2]. Strassen's algorithm for matrix multiplication and improvements later can be found in [3, 4]. The stable matching problem is studied in [5, 6]. The Clay Mathematics Institutes listed seven fundamental mathematic problems as the Millennium Prize problems, which are "important classic questions that have resisted solution for many years." [7]. One of them is the P vs. NP problem.

References

1. Kernighan BW (2017) Understanding the digital world: what you need to know about computers, the internet, privacy, and security. Princeton University Press, Princeton
2. Knuth DE (1997) The art of computer programming. Addison-Wesley, Boston
3. Strassen V (1969) Gaussian elimination is not optimal. Numer Math 13:354–356
4. Karstadt E, Schwartz O (2017) Matrix multiplication, a little faster. In: Proceedings of the 29th ACM symposium on parallelism in algorithms and architectures, pp 101–110
5. Gale D, Shapley LS (1962) College admissions and the stability of marriage. Am Math Monthly 69(1):9–15
6. Chen J, Skowron P, Sorge M (2019) Matchings under preferences: Strength of stability and trade-offs. In: Proceedings of the 2019 ACM conference on economics and computation, pp 41–59
7. https://www.claymath.org/millennium-problems

Chapter 5
Systems Thinking

Enable people to follow the way, without them having to understand [the internals of] it.
—Confucius (551–479 BCE)
Inside every large program, there is ***an algorithm*** *trying to get out. [Here, algorithm also means specification or high-level design of a system]*
—Leslie Lamport, 2018

Computational processes execute on computing systems, including computer systems and computing application systems. Systems thinking is the way of thinking to make computational processes **practical**. Systems thinking must systematically and thoroughly address all necessary details and complexities. That is,

$$\begin{aligned}\text{Being practical} &= \text{Being thorough} + \text{Being systematic}\\ &\quad + \text{Coping with complexity}.\end{aligned}$$

Without systems thinking, we would not have had the vibrant computing ecosystem today, with billions of users using millions of applications on various devices.

The main character of systems thinking is: using **abstractions** to compose **modules** into a system**,** to enable **seamless execution** of computational processes. This chapter discusses systems thinking with its three key concepts: abstraction, modularization, and seamless transition.

Abstraction is a creative process of abstracting the essentials, as well as the process's outcome. Computer science abstractions mainly consist of **data abstractions** and **control abstractions**. All abstractions have three properties: constrained, objective, and generalizable, i.e., the **COG** properties.

Modules are a special type of abstractions which enforce the **information hiding** principle. A system is comprised of modules by interconnecting their interfaces. We discuss a number of hardware and software abstractions. The discussions are ordered by levels of abstractions, from the lowest-level logic gates to the highest-level

Z. Xu, J. Zhang, *Computational Thinking: A Perspective on Computer Science*,
https://doi.org/10.1007/978-981-16-3848-0_5

application software, including combinational circuits, sequential circuits, instruction pipeline, von Neumann architecture, and software stack.

Seamless execution is concerned with how to ensure smooth executions of computational processes. What is the first instruction to execute? Where to find the next instruction? How to transition from one instruction to the next? Is there any bottleneck? What if there is abnormality? We discuss four “laws” which make seamless execution possible: Yang’s cycle principle, Postel’s robustness principle, von Neumann’s exhaustiveness principle, and Amdahl’s law.

5.1 Systems Thinking Has Three Objectives

Let us consider computing without systems thinking, even having the benefits of logic thinking and algorithmic thinking. We can still theoretically solve all computable problems, by executing algorithms on Turing machines. But we quickly run into practical problems, as illustrated by the following example.

Example 5.1. Understand Message Storage in WeChat
The WeChat (微信) application system is a popular computer application service. In 2018, the WeChat community exceeded 1 billion users worldwide. Can we understand or design a WeChat system by executing algorithms on Turing machines? It is not practical to do so.

Let us consider a specific problem in the WeChat system, the **message storage problem**: when I send a chat message to my family, where should the WeChat service store my message? This storage problem is not a logic problem or an algorithmic problem. We cannot easily formulate and solve the storage problem and evaluate alternative solutions, as we do for the sorting problem. It is a systems problem involving thoughtful considerations and tradeoffs involving many issues of practical systems. A way of systems thinking is needed, which has three main objectives.

- Being **thorough**. The WeChat system design considers all necessary issues such as functionality, user experienced convenience, performance, fault tolerance, privacy, as well as system scalability.
- Being **systematic**. WeChat adopts a systematic approach called cloud computing. Consequently, messages are stored on the WeChat cloud datacenter.
- **Coping with complexity**. There are many possibilities to consider. A message can be stored on many places: my smartphone, my family members’ devices, the WeChat system, or a third-party platform. Which one is the correct or the better choice? How to go about answer such a question?

☷

Systems thinking strives to simultaneously achieve all three objectives. This makes system thinking a synergy of science, engineering, and art. It is the reason why a system designer is often known as an *architect*.

5.1.1 Being Thorough

In understanding or designing a system such as WeChat, we need a *thorough* way to cover all the details of the integrated whole system from end to end, ignoring no necessary details.

For instance, when considering where to store a message, we need to consider the entire path the message may traverse, from the message sender end to the message receiver end. We also need to consider the whole stack of systems components from high-level user interfaces, algorithmic descriptions, software and hardware, down to the lowest level of automatic execution on a computer.

We use three examples to illustrate how systems thinking emphasizes thoroughness. One example is representative of necessary but boring details. The other two illustrate how to take care of necessary details by using clever abstractions. Systems thinking requires that all necessary details should be considered, even those boring ones. At the same time, systems thinking strives to avoid simple-minded enumeration of all details, by using abstractions.

Example 5.2. Big Endian Versus Little Endian Data Representations
A necessary but boring detail is the ordering of bytes of a multi-byte number. We do not see such details in the design and analysis of an algorithm.

For instance, a 32-bit integer 1078018627_{10} can also be written as 0x40414243, which consists of four bytes, where Byte0 is 01000000=0x40, Byte1 is 01000001=0x41, Byte2 is 01000010=0x42, and Byte3 is 01000011=0x43. Most computers have a byte addressable memory. When storing this number in memory, we need four consecutive memory byte cells starting at address A.

The problem is: when storing this number in memory, in what order to place the four bytes 0x40, 0x41, 0x42, 0x43 in the four memory cells A, A+1, A+2, A+3? Two representations (orderings) are used in practice, as shown in Fig. 5.1.

The **little endian** ordering places the least significant byte, i.e., 0x43 of 0x40414243, in the smallest address A, and the most significant byte, i.e., 0x40 of 0x40414243, in the biggest address A+3. The **big endian** ordering simply reverses

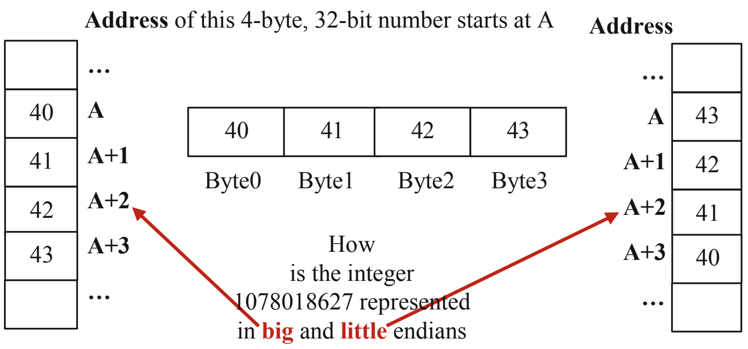

Address of this 4-byte, 32-bit number starts at A
Address
…
40
A
41
A+1
42
A+2
43
A+3
…
40
41
42
43
Byte0
Byte1
Byte2
Byte3
How is the integer 1078018627 represented in big and little endians
A
43
A+1
42
A+2
41
A+3
40

Fig. 5.1 Big endian versus little endian representations of integer 1078018627

the order. Similarly, in computer communication, where a bit-stream is transmitted between two computers, the little endian ordering sends the least significant bit first, and the big endian ordering sends the most significant bit first.

The big endian versus little endian ordering issue got its name from *Gulliver's Travels*, a satirical novel by Jonathan Swift. In a fictional country Lilliput, two factions fought a holy war over from which end to break a boiled egg. In 1980, Danny Cohen, at Information Sciences Institute, University of Southern California, applied the terms big endian and little endian to computer science. Cohen holds the viewpoint that "Agreement upon an order is more important than the order agreed upon." Forty years later, the computer science fields settled into a situation where different products and communities use different endians within themselves, and cross-community applications convert the data format whenever necessary. Some example communities follow:

- Big endian: TCP/IP networks, MIPS processors
- Little endian: x86 processors, ARM processors, RISC-V processors

Example 5.3. A Single Abstraction for Millions of I/O Devices
Recall that the von Neumann model of computer has three families of components: the processors, the memory units, and the I/O devices. Of these three, the family of I/O devices is the largest family, with over a million different types of devices having been built and used.

The problem is: how do millions of applications deal with so many I/O devices?

For instance, a scratch pad is quite different from a keyboard, although both are input devices. A display monitor is quite different from a printer, although both are output devices.

We must be thorough, allowing a computer to interact with any I/O devices. A brute-force approach is to design a method for an application to interact with each device. This is not feasible, since there are million $\times$ million combinations.

Computer scientists have come up with an ingenious abstraction, called *device driver*, as illustrated in Fig. 5.2.

Applications only need to interact with the generic device driver interface, which is quite similar to the interface of accessing files. Applications only see two generic types of I/O devices: *block devices* and *character devices*. A computer application interacts with a character device one character at a time, such as printing out a program output on the display monitor in command-line mode, one character at a time. Interactions with a block device allow us to input or output a larger chunk of information (called a *block*) at a time, such as outputting an image on the display monitor in graphic mode.

When an I/O device product, say a new scratch pad called Device1, is developed, the device vendor also develops a unit of software, say Driver1, which is the device driver for Device1. This device driver software is developed, say, with 1 man-month cost. It implements the generic interface and realizes all particulars of detailed I/O operations, hiding all such details from applications.

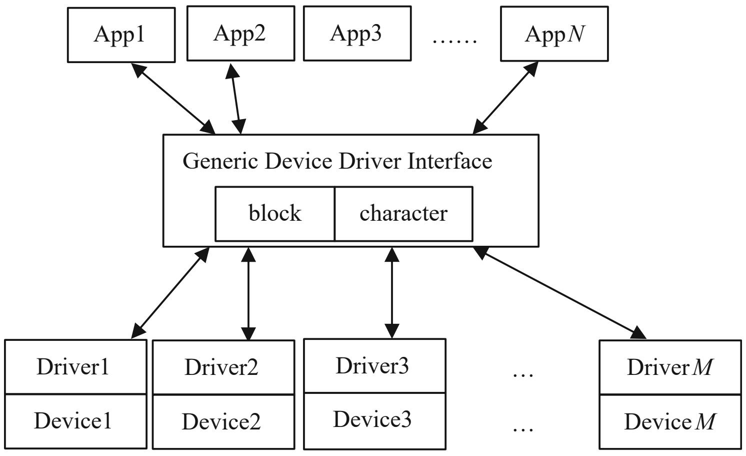


Fig. 5.2 Illustration of the device driver abstraction

Note that such a device driver abstraction drastically decreases the development cost, from $M \times N$ to M+N where M is number of different applications and N is the number of different devices. Since N and M are numbered in millions, the total cost is reduced from trillions to millions of man-months. For each device, the development cost is reduced from millions of man-months to a few.

The above example shows that in achieving thoroughness, systems thinking avoids painful enumeration of all details, by using clever abstractions. We consider another case to further illustrate this point.

Example 5.4. Measure Supercomputers by Smartly Designed Benchmarks
Supercomputers are the fastest computers in the world. We want supercomputers to increase their speed a thousand-fold every 10 years. Here lies a basic problem: How do we precisely define and measure this objective? This problem can be further divided into two more detailed questions:

- How to measure the speed of a supercomputer?
- How to be thorough, i.e., consider all applications when measuring the speed?

An answer to the first question is straightforward: run a small set of representative application programs, called **benchmarks**, and count the number of operations executed per second. Supercomputers measure speed with **FLOPS**, for 64-bit floating-point addition and multiplication operations executed per second.

The second question is more difficult. What are *representative* benchmark programs? A benchmark suite usually consists of no more than a dozen programs. How can it represent the thousands of supercomputing applications today? Systems thinking suggests a method shown in Fig. 5.3. Supercomputing speed is significantly influenced by a phenomenon called locality. **Temporal locality** refers to the fact that

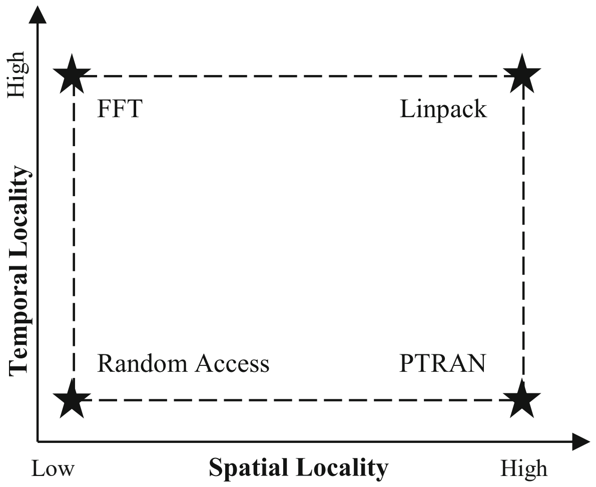


Fig. 5.3 Illustration of using benchmarks of various representative localities

data and instructions currently used tend to be used again in near future. **Spatial locality** refers to a similar fact regarding address space.

The supercomputing community designed a benchmark suite containing four benchmark programs, representing the four extreme combinations of temporal and spatial localities: low-low, low-high, high-low, and high-high. Any other application falls within the area enclosed by the dashed lines.

5.1.2 *Being Systematic*

We have literally thousands of types of computers and millions of computer applications today. It is unthinkable to design or understand each of them in an arbitrary, ad hoc way. Fortunately, computer science has created a systematic, layered approach to support millions of applications for billions of users on their favorite computers. This approach is called the technology **stack** approach. Figure 5.4 shows a hardware-software stack for a desktop PC. Similar stacks are available for embedded computers, smartphones, servers, and supercomputers.

In Fig. 5.4, the colored parts represent hardware components of a computer. These components are formed from logic gates, combinational circuits, and sequential circuits, which in turn are constructed from transistors and wires, as well as capacitors and resistors. Magnetic parts are the main components of hard disks. There are millions of products of I/O devices.

The upper three layers of the stack are software. Instructions are the smallest units of software. System software (such as operating system and compiler) and application software are made of instructions. When any application program starts to execute, a **process** of the application is created. Thus, a process is a program in

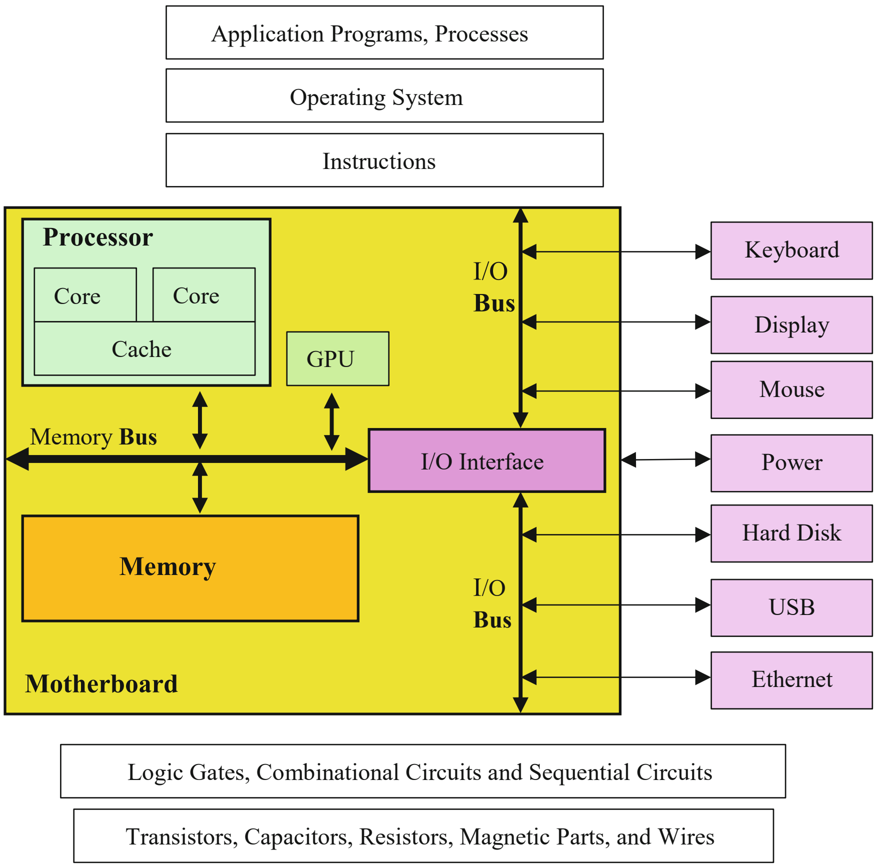


Fig. 5.4 Illustration of the hardware-software stack of a desktop personal computer

execution. The operating system manages the processes by scheduling them to execute on the computer hardware at certain times.

The stack of layers in Fig. 5.4 conveys two meanings of being systematic.

- First, the same stack is used to support millions of applications, instead of a million stacks, each for one application.
- Second, an upper layer provides a higher level of abstraction than the lower layers, and the upper layer abstraction utilizes primitive resources or abstractions from the lower layers. For instance, a processor is made of combinational circuits and sequential circuits, which in turn are made of transistors and wires. A processor has a much higher level of abstraction than transistors and wires.

5.1.3 Coping with Complexity

Complexity here refers to systems complexity, which is different from algorithmic complexity in Chap. 4. A system is either a computer system, such as a laptop computer, or a computer application system, such as WeChat. Complexity makes it difficult to understand, design, and use a computing system. Systems thinking is used to cope with complexity. We have already seen millions of practically working computing systems, including WeChat. These successful cases testify that systems thinking provides effective supports for coping with complexity.

Many factors contribute to systems complexity. We discuss four such factors which appear frequently.

The first factor is **system scale**, or system size, which refers to the number of components of a system. For instance, the main processor microchip in a smartphone has over 2 billion transistors. If we liken the microchip to Planet Earth, and compare transistors and wires to buildings and roads, the transistor layout diagram of a microchip is as complex as a country's meter-scale map, showing all buildings and roads. The hardware system used by WeChat is as complex as a meter-scale world map. We cannot understand such a complex system as a set of 2 quintillion (billion billion) transistors.

The second factor is **system heterogeneity**, or diversity, which refers to the number of different types of components in a system. For instance, WeChat has over 1 billion users worldwide, where each user uses one or more computing devices to access the WeChat services. These devices have much heterogeneity and diversity. A device could be a smartphone, a pad, a PC, or even a robot on a server. The smartphones are made by thousands of different companies. It is a wonder that WeChat works at all with such device heterogeneity and diversity.

The third factor is **system organization**, i.e., how the components are connected and organized into a system. A haphazard mess of interconnected components is much more complex than a system with clear, principled organization.

The fourth factor is **system variation**, which refers to the fact that the system or its components are often not stable, but keep changing. Furthermore, they may change at different times and different rates. Sometimes, we also call system variation as system dynamicity. Examples of such changes include: deploying new products or services, upgrading an existing service or product, bug-fixing, etc.

In summary, to make computing practical, we need systems thinking which is thorough, systematic, and can cope with systems complexity. Computer science has accumulated a rich body of knowledge for systems thinking. The essence is using **abstractions** to compose **modules** into a system, to enable **seamless execution** of computational processes. In other words, systems thinking has three mental tools: abstraction, modularization, and seamless transition. The beauty of systems thinking is that even with seemingly impossible thoroughness and systematic requirements, these mental tools enable us to cope with such complexity presented in today's computer application systems consisting of billions of diverse users and devices.

5.2 Abstraction

Abstraction is the creative process of abstracting a high-level concept from low-level instances, which are full of irrelevant details and particularities. An abstraction is also the outcome of the creative process of abstracting. This book focuses on computing abstractions, namely, the abstracting process that produces abstractions of information transformation by digital symbol manipulations. From the systems viewpoint, this book studies four classes of abstractions, as listed in Table 5.1. They are (1) data representations, (2) software abstractions, (3) hardware abstractions, and (4) the von Neumann architecture model that bridges software and hardware.

5.2.1 Three Properties of Abstraction: COG

All abstractions have three properties, called the **COG properties** of abstraction. That is, any computing abstraction is constrained, objective, and generalizable.

- **C**onstrained. An abstraction is a high-level concept specification constrained by hiding details. Abstraction focuses on the essential aspect when specifying the computing system (or a subsystem) from one perspective, while hiding or ignoring details from other perspectives and particularities of individual instances. The ability to hide and ignore, namely, to constrain, is why abstraction can cope with complexity.

Table 5.1 Four classes of abstractions

Data type	bit (1 bit), hexadecimal number (4 bits), byte (8 bits), uint8 (8-bit unsigned integer), integer (64 bits); array (n elements of the same type), slice (a descriptor pointing to an array); text file, BMP image file; hypertext and hyperlink	
Software	Algorithm	Smart method of information transformation, such as quicksort, hiding text in a BMP file, etc.
	Program	Code realizing algorithms in computer language, such as hide.go in the Text Hider project
	Process	Program in execution, such as the “hide” process running in a Linux environment
	Instruction	The smallest unit of software, directly executable by computer hardware
von Neumann Architecture: a computer model bridging software and hardware		
Hardware	Instruction Pipeline	The basic hardware mechanism to automatically execute any instruction
	Sequential Circuit	More precisely, only consider Synchronous Sequential Circuit comprised of combinational circuits and state circuits and driven by a clock signal; equivalent to the automata concept
	Combinational Circuit	Aka Boolean circuit, realizing a Boolean function

- **O**bjective. Abstraction does not imply up-in-the-air vagueness or ambiguity. A computing abstraction is a named, objective entity. It is a precisely defined concept, both syntactically and semantically, and cannot be arbitrarily interpreted or changed by any particular human's whim. Objectivity makes computing abstractions bit-accurate and automatically executable.
- **G**eneralizable. An abstraction should be generalizable to unseen instances or unexpected scenarios. Computing abstraction is created by humans. The created abstraction should be able to handle existing abstractions or instances already seen, as well as unseen instances and unexpected scenarios. This capability of generalization is why we can use one set of abstractions of concepts and methods to solve all problems encountered, instead of treating each individual problem instance individually.

Example 5.5.The COG Properties of Unicode
The 128-character ASCII set is sufficient to encode English text. Can we do the same for all the world's text? The computer science community has come up with a beautiful abstraction called **Unicode** to solve the problem. By the year 2020, Unicode already encodes over 143,000 characters in 154 languages, including over 70,000 Chinese characters.

"Encoding the world's writing systems by a number of bits" is the process of abstracting. The Unicode abstraction is the outcome of this abstracting process. It has three properties of COG.

- Unicode is **constrained**. It focuses on one essential task: encoding the world's writing systems, or character sets. It ignores issues such as the font, the size, the alignment of the character, whether it is boldface or italic, etc.
- Unicode is **objective**. Unicode is precisely defined. The Chinese character '志' and the Euro sign '€' have encodings U+5FD7 and U+20AC, respectively, without ambiguity.
- Unicode is **generalizable**. Unicode uses a standard abstraction to solve the problem of "encoding text in the world's writing systems". It is not tied to any computer hardware, software, or application scenario. It is used on our PCs, smartphones, and the World Wide Web. Most computer devices and applications supported by Unicode did not exist when Unicode was designed.

5.2.2 Data Abstractions

Computing abstractions mainly manifest as data abstractions and control abstractions. **Data abstractions**, also called **data types**, are abstractions of data, including operations on such data. Examples of data abstractions include number systems, bit, byte, character, string, integer, floating-point number, array, slice, text file, BMP image file, hypertext and hyperlink. **Control abstractions** are abstractions specifying the control structure of a system, that is, when and how to invoke which parts of a system, to give order to the totality of the system. Examples of control abstractions include operator precedence, sequence, selection, iteration, and function. Some abstractions exhibit features of both data abstractions and control abstractions. We discuss several examples in this section. Section 5.3 contains more examples of abstractions as hardware and software modules.

5.2.2.1 Positional Notation of Number Systems

Most modern number systems are **positional number systems**. That is, they use a positional notation to represent a number such that the value of the number is determined by its digits and the position of each digit. The following example shows why the positional notation is popular: it makes doing arithmetic easy, as we do with the usual decimal notation.

Example 5.6. Find a Person's Age Using Roman and Decimal Numerals
A person was born in the year MCMLIV. What's his age in the year MMXXI?

The above question uses Roman numerals. We need to find out

MMXXI – MCMLIV = ?

Note the following mapping between Roman numbers and decimal numbers.

Roman	M	D	C	L	X	V	I	IV	IX	XL	XC	CD	CM
Decimal	1000	500	100	50	10	5	1	4	9	40	90	400	900

Using Roman numerals to do arithmetic is difficult. Converting the numbers into the decimal notation makes it easier: $2021 - 1954 = 67$. The person is 67 years old, or LXVII years old in Roman numerals. That is, MMXXI – MCMLIV = LXVII.

Why is the decimal notation so much easier? Because it is a **positional notation**, as illustrated in Fig. 5.5. The decimal number $a = a_3a_2a_1a_0.a_{-1}a_{-2}a_{-3}a_{-4} = 2021.1954$ has eight decimal **digits**. Two digits have identical symbol 2. But, because of their different positions, they represent different values. The leftmost 2 represents two thousands, and the second 2 represents two tens. Note that the value of a is $a = \sum_{i=-4}^{3} a_i \times 10^i$, where i is the **index**.

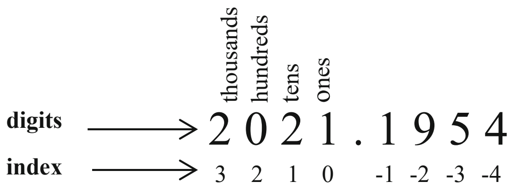


Fig. 5.5 Explaining the positional notation of decimal number 2021.1954

Example 5.7. Other Positional Number Systems

In general, the value of a natural number a represented in a base-b, n-digit positional notation is evaluated by

$$a = \sum\nolimits_{i=0}^{n-1} a_i \times b^i$$

where digit a_i takes a value from the **digit set** {0, 1, . . ., b-1}.

For binary notation, we have

$$a = \sum\nolimits_{i=0}^{n-1} a_i \times 2^i$$

where digit a_i takes a value in the digit set {0, 1}.

With a hexadecimal positional notation, we have

$$a = \sum\nolimits_{i=0}^{n-1} a_i \times 16^i$$

where digit a_i takes a value in the digit set {0, 1, . . ., 15}={0, 1, . . ., F}.

It is apparent that a particular positional number system is determined by three things: its word-length n, its base b, and its digit set.

To more clearly understand the concept of positional number system, let us ask some questions: What values are allowed for the base and the digit set? What is the relation between the base and the digit set? It appears that the base should be a positive integer k, and the digit set is {0, 1, . . ., k-1}. We have seen the cases for $k =$ 2 (binary), 10 (decimal), and 16 (hexadecimal).

Can we have radically different positional number systems? Can we use Fibonacci numbers? Can we use an irrational base?

The answers are YES. In 1957, George Bergman, then a 12-year junior high school student, proposed to use {0, 1} as the digit set and the Golden ratio $\tau = (1+\sqrt{5})/2 \approx 1.6180339$ as the base. He also showed how to do arithmetic in this τ *number system*. Later work on Fibonacci Number System (FNS) showed how to represent numbers using 0 and 1 as digits and Fibonacci numbers as positional weights.

Table 5.2 contrasts these number systems with the more familiar decimal, hexadecimal, and binary number systems. For instance, the decimal number 14 has hexadecimal and binary representations of E and 1110, respectively. With the τ number system, 14 is represented as the following:

$$14 = \mathbf{100100.110110} = \tau^5 + \tau^2 + \tau^{-1} + \tau^{-2} + \tau^{-4} + \tau^{-5}.$$

With the Fibonacci Number System (FNS), 14 is represented as follows:

Table 5.2 Decimal, hexadecimal, binary, tau, and Fibonacci number system representations

Decimal	Hexadecimal	Binary	The τ number system	FNS
$10^1 10^0$	16^0	$2^3 2^2 2^1 2^0$	$\tau^5\tau^4\tau^3\tau^2\tau^1\tau^0\tau^{-1}\tau^{-2}\tau^{-3}\tau^{-4}\tau^{-5}\tau^{-6}$	8 5 3 2 1
0	0	0000	0	00000
1	1	0001	1	00001
2	2	0010	10.01	00010
3	3	0011	100.01	00100
4	4	0100	101.01	00101
5	5	0101	1000.1001	01000
6	6	0110	1010.0001	01001
7	7	0111	10000.0001	01010
8	8	1000	10001.0001	10000
9	9	1001	10010.0101	10001
10	A	1010	10100.0101	10010
11	B	1011	10101.0101	10100
12	C	1100	100000. 101001	10101
13	D	1101	100010.001001	11000
14	**E**	**1110**	**100100.110110**	**11001**
15	F	1111	100101.001001	11010

$$14 = \mathbf{11001} = 1 \times 8 + 1 \times 5 + 0 \times 3 + 0 \times 2 + 1 \times 1.$$

5.2.2.2 Representing Real Numbers

From binary-decimal number conversion in Sect. 2.1, we already know that a real number can be represented by using a pair of numbers, for the whole part and the fraction part, respectively. For instance, $\pi \approx 3.1415927$ can be represented in binary as 11.0010010000111111011011, after decimal-to-binary conversion.

What happens in reality is more sophisticated. Real numbers are represented in computers as **floating-point numbers**. Similar to the *scientific notation* of numbers, they use two fixed-point numbers for the exponent and the significant parts, respectively. For instance, a possible floating-point representation for π is

$$\pi \approx 31415927 \times 10^{-7}$$

where -7 is the exponent and 31415927 is the significant (also called mantissa).

The most widely used floating-point number representations are the *IEEE 754 floating-point standard*. The IEEE 754 32-bit format uses one bit (the leftmost bit) for the sign, 8 bits for the exponent, and 23 bits for the significant. It can represent $\pi \approx 3.1415927$ with a precision up to the seventh digit after the decimal point. The IEEE 754 64-bit format uses one bit for the sign, 11 bits for the exponent,

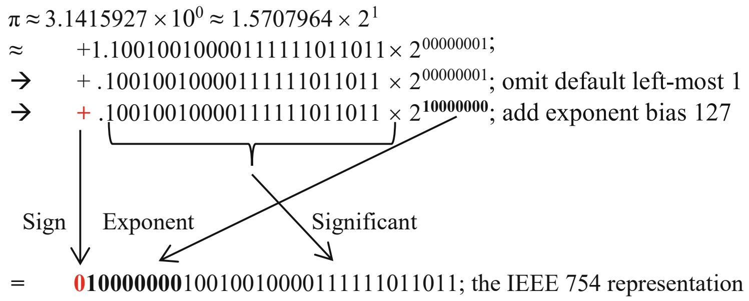

π ≈ 3.1415927 × 10^0 ≈ 1.5707964 × 2^1
≈ +1.10010010000111111011011 × 2^00000001;
→ + .10010010000111111011011 × 2^00000001; omit default left-most 1
→ + .10010010000111111011011 × 2^10000000; add exponent bias 127
Sign
Exponent
Significant
= 01000000010010010000111111011011; the IEEE 754 representation

Fig. 5.6 Representation of π in IEEE 754 32-bit floating-point standard

and 52 bits for the significant. It can represent $\pi \approx 3.141592653589793$ with a precision up to the 15th digit after the decimal point. The 64-bit format doubles the precision of the 32-bit format. Thus, the IEEE 754 32-bit format is also called the single-precision format, and the 64-bit format called the double-precision format.

Let us look at the representation $\pi \approx 31415927 \times 10^{-7}$ more carefully. This number could also be represented as $\pi \approx 3.1415927 \times 10^0$ or $\pi \approx 0.31415927 \times 10^1$. Such multiple representations of the same value may cause confusion.

We can achieve representation uniqueness with **normalized significant**, i.e., placing the binary point to the right of the leftmost non-zero bit of the significant. Thus, π is uniquely represented as $\pi \approx 1.10010010000111111011011 \times 2^{00000001}$ in binary notation, or $\pi \approx 1.5707964 \times 2^1$. Since the left-most bit is always 1 for any non-zero numbers, it can be omitted to save one bit of representation.

The IEEE 754 standard also uses **biased exponent** to enable fast exponent comparison operation. That is, an exponent bias of 127 is added to the exponent value for 32-bit representation, which is subtracted when interpreting a number's value. Thus, the exponent 00000001 becomes 1+127=128, or 10000000 in binary notation. The final IEEE 754 32-bit representation for π is shown in Fig. 5.6. Similarly, the 64-bit representation for π is

01000000000010010010000111111011010101000100010000101101000
11000.

Besides normalized significant and biased exponent, the standard has several additional clever designs, especially in representing and handling exceptions and errors. Besides normal values, the IEEE 754 standard also includes representations for not so normal floating-point numbers, such as positive and negative infinities ($\pm\infty$), subnormal numbers (representing underflowing values), and Not a Number values (NaNs, such as trying to find $\sqrt{-5}$).

All the above systematic thinking helps ensure the algebraically completeness of floating-point arithmetic, such that many arithmetic errors, one of which caused the loss of an Ariane 5 rocket in 1996, could be avoided. Being a sophisticated abstraction, IEEE 754 is widely used in smartphones, laptops, servers to supercomputers. Professor William Kahan was awarded the Turing Award for his fundamental contributions to numerical analysis, such as embodied in IEEE 754.

5.2.2.3 Test If Two Floating-Point Numbers Are Equal

Table 1.2 already hints that in general, computer representations of integers and characters are exact, but computer representations of real numbers, i.e., floating-point numbers, are often approximate. Any computer has finite word lengths and finite memory capacity. It cannot hold the exact value of a digital symbol that requires an infinite number of bits, such as some real numbers.

The program in Fig. 5.7 generates the following seemingly inconsistent outputs:

```
> go run ./testPoint123.go
0.1+0.2 == 0.3
0.1+0.2 != 0.3
0.1+0.2 == 0.3
>
```

The code shows that it is a wrong way to use the double-equal operator "==" to compare two floating-point expressions. Instead, we should compare the absolute difference against a small threshold value, e.g., $|(X + Y) - Z| < 10^{-12}$.

```
package main
import "fmt"
import "math"
func main() {
        if 0.1 + 0.2 == 0.3 {
                fmt.Println("0.1+0.2 == 0.3")
        } else {
                fmt.Println("0.1+0.2 != 0.3")
        }
        X := 0.1            // var X float64 = 0.1
        Y := 0.2
        Z := 0.3
        if X + Y == Z {
                fmt.Println("0.1+0.2 == 0.3")
        } else {
                fmt.Println("0.1+0.2 != 0.3")
        }
        if math.Abs(X+Y - Z) < math.Pow(10, -12) {
                fmt.Println("0.1+0.2 == 0.3")
        } else {
                fmt.Println("0.1+0.2 != 0.3")
        }
}
```

Fig. 5.7 Code testPoint123.go illustrating the strange phenomenon: 0.1+0.2 != 0.3

5.2.2.4 ASCII, Unicode, and UTF-8

ASCII is a US standard for encoding English text, covering 128 symbols. Unicode is an international standard that encodes over 143,000 characters in 154 languages, including over 70,000 Chinese characters, also known as CJK Unified Ideographs. Unicode also encodes other symbols, such as format characters, control characters and emoji symbols. A dominant implementation of Unicode is **UTF-8** (8-bit Unicode Transformation Format), which is capable of encoding all characters in Unicode. By January 2020, 94.6% of character encodings for the websites worldwide use UTF-8, while fewer than 0.1% use ASCII.

UTF-8 uses one to four bytes when encoding different character sets. The first byte of UTF-8 (0X00 to 0X7F to be exact) has the same encodings as ASCII, as shown in Table 5.3. Thus, only one byte is needed for an ASCII character. The UTF-8 values for the 128 ASCII characters range from 0x00 for the NUL character to 0X7F for the DEL character. On the other hand, a Gothic character needs four bytes. Three bytes are needed for a Chinese character, also called Hanzi. For instance, the Chinese character 志 has a Unicode value of U+5FD7, and a UTF-8 value of 0XE5BF97 (3 bytes for E5 BF 97). For the Euro sign "€", the Unicode value is U+20AC and the UTF-8 value is 0XE282AC. For the Greek capital letter Omega "Ω", the Unicode value is U+03A9 and the UTF-8 value is 0XCEA9.

5.2.2.5 Review of Bit, Byte, Character, Integer, Array, and Slice

In Chaps. 1 and 2, we introduced the rudiment concepts of six data types: bit, byte, character, integer, array, and slice. Here we put these data abstractions together in one place, to better see their distinctions and differences. We pay special attention to how these data abstractions are stored in memory and how they are represented when printing out. The six data types are illustrated in Fig. 5.8.

A quick way to understand these data types is to execute the following program.

```
X := byte(63)          // X is a byte variable
fmt.Printf("Decimal: %d\n", X)     // Decimal: 63
fmt.Printf("Hex: %X\n", X)     // Hex: 3F
fmt.Printf("Character: %c\n", X)     // Character: ?
```

Table 5.3 ASCII, Unicode, and UTF-8 representations of five typical characters

Symbol	Description	ASCII	Unicode	UTF-8	Bytes needed by UTF-8
T	English capital letter T	0X54	U+0054	0X54	1
Ω	Greek letter Omega	N/A	U+03A9	0XCEA9	2
€	The Euro sign	N/A	U+20AC	0XE282AC	3
志	A Chinese character	N/A	U+5FD7	0XE5BF97	3
⊙	A Gothic letter	N/A	U +10348	0XF0908D88	4

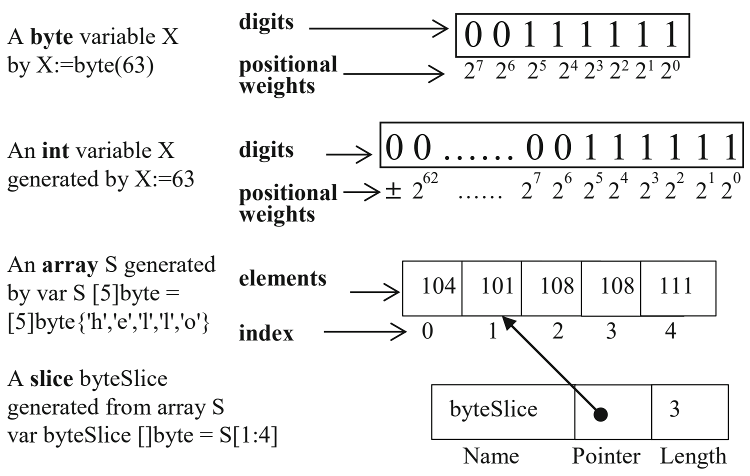


Fig. 5.8 Differences among byte, int, array, and slice

```
   fmt.Printf("Binary: %b\n", X)     // Binary: 111111
   var S [5]byte = [5]byte{'h','e','l','l','o'}  // S=[104, 101, 108,
108, 111]
   var byteSlice []byte = S[1:4]     // byteSlice=[101, 108, 108]
   fmt.Println("array S = ", S)   // Array S = [104 101 108 108 111]
   fmt.Println("byteSlice = ", byteSlice)  // byteSlice = [101 108 108]
```

Try to execute the same code and see what happens when “X = byte(63)” is replaced by “X = 63” and “X = 8364”. The meanings of variables for byte, int and array are straightforward. The left-most bit is the sign bit in the int type.

The slice variable byteSlice is a data structure describing a section of an underlying array (S[1], S[2], S[3]), i.e., the array section S[1:4] starting at index 1 and has a length of 3. Note that S[4] is not part of S[1:4].

Slices can also be created with the built-in **make** function. The function call

```
   make([]int, n+1)
```

allocates an array of length n+1 and returns a slice that refers to that array. All elements of the array are type int and initially set to zero.

Note that we only use four data types in the Go language practices, i.e., byte, int, array and slice. So how to treat character and bit? Two practices follow.

- A character such as the question mark ‘?’ is represented as a byte value 00111111, which is equivalent to an 8-bit unsigned integer 00111111=63. The two data types **byte** and **uint8** are equivalent.
- To operate on a bit, we can operate on a byte or an integer containing that bit.

Example 5.8. Inverting the Least Significant Bit of a Byte
Suppose we want to invert the least significant bit (the right-most bit) of a byte 0011111**1**, to obtain 0011111**0**. We cannot directly do it with one operation but need a sequence of operations such as the following:

```
x := byte(63)   // assign 63₁₀=00111111₂ to variable x
v := ^x     // bitwise NOT of x, i.e., v=11000000
v = v & 0x1   // bitwise AND to retain the right-most bit of v, i.e.,
      v= 11000000 & 00000001
x = x & 0xFC   // bitwise AND to clear the right-most bit of x, i.e.,
      x= 00111111 & 11111110
x = x | v     // bitwise OR to invert the last bit of x, i.e.,
      x=00111110 | 00000000=00111110
```

Let us use a sequence of equations to show how the above code works. To invert the least significant bit of 0011111**1**, the code executes as follows:

$x = 0011111\mathbf{1}$	Given input
$v = \overline{0}\,\overline{0}\,\overline{1}\,\overline{1}\,\overline{1}\,\overline{1}\,\overline{1}\,\overline{1} = 1100000\mathbf{0}$	Bitwise NOT
$v = 11000000 \,\&\, 00000001 = 0000000\mathbf{0}$	Bitwise AND
$x = 00111111 \,\&\, 11111110 = 0011111\mathbf{0}$	Bitwise AND
$x = 00111110 \mid 00000000 = 0011111\mathbf{0}$	Bitwise OR

Example 5.9. Replacing the Least Significant 2 Bits of a Byte
Suppose we want to replace the least significant 2 bits of a byte 001111**11**, with the least significant 2 bits of another byte 001010**10,** to obtain 001111**10**. We can realize this with the following sequence of operations:

```
x := byte(63)   // assign 63₁₀=00111111₂ to variable x
v := byte(42)   // assign 42₁₀=00101010₂ to variable v
v = v & 0x3   // bitwise AND to retain the right-most 2 bits of v, i.e.,
      v= 00101010 & 00000011 = 00000010
x = x & 0xFE   // bitwise AND to clear the right-most 2 bits of x, i.e.,
      x= 00111111 & 11111100 = 00111100
x = x | v     // bitwise OR to replace the last 2 bits of x, i.e.,
      x=00111100 | 00000010 = 00111110
```

The above code replaces the least significant 2 bits of variable x with the least significant 2 bits of variable v. This becomes clearer with the following sequence of equations to show how the above code works step by step.

```
x = 00111111            Given input
v = 00101010            Given input
v = 00101010 & 00000011 = 00000010     Bitwise AND
x = 00111111 & 11111100 = 00111100     Bitwise AND
x = 00111100 | 00000010 = 00111110     Bitwise OR
```

5.2.2.6 Pointers and Addressing Modes

The array data type, such as **var a [100]int**, has an obvious advantage: it is a simple linear arrangement of 100 integers consecutively stored in memory, and an array element a[i] can be referenced by index i. Computer scientists realized a fact about data structure and data layout in memory: although the linear arrangement of an array is easy to understand and operate, allowing nonlinear arrangements, i.e., the elements can jump around, brings flexibility.

Such nonlinear arrangements are realized by a basic mechanism called **pointers**, which is implemented by the **indirect addressing** mode provided by hardware. A pointer holds the *address* of a value, not the value itself.

We already encountered several addressing modes, such as base+index+offset in Chap. 2. Three addressing modes are compared below:

- Immediate mode:
  MOV 50, R1; assign the immediate value 50 to R1
- Direct mode:

  MOV M[50], R1; assign M[50] to R1
- Indirect mode:

  MOV M[M[50]], R1; assign M[M[50]] to R1

Registers are regarded as special memory cells.

In a high-level language such as Go, an ordinary variable holds a value, such as an integer or a byte value. A pointer variable holds the address of a variable.

Example 5.10. Contrasting a Pointer Variable to an Ordinary Variable
Figure 5.9 contrasts a normal variable b and pointer variable p.

Any variable has three attributes: a name, a data type, and a value. When a program is compiled to execute, the variable name is bound to a memory address, called the variable's address, which can be obtained with the '&' operator.

For instance, after the following declaration statement

```
var b bool = true
```

we have a variable named b of Boolean type, with an initial value of true.

The asterisk '*' symbol is used in a declaration statement to declare a pointer variable:

```
var p *bool = &b
```

Here, the pointer variable is named p, which points to a value of Boolean type, with an initial value as the address of b, which happens to be 0xc042058058.

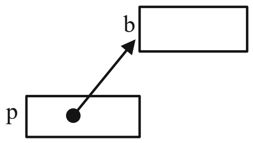
The asterisk '*' symbol is also used in an expression as a **dereference** operator, to obtain the value at the address pointed to by a pointer variable.

```
package main
import "fmt"
func main() {
  b := true          // Boolean variable b
  p := & b           // p holds b’s address
  fmt.Println(p)     // Print b's address
  fmt.Println(*p)    // Print b's value
  *p =  false        // Modify b's value
  fmt.Println(b)     // Print b's value
  *p = !(*p)         // Use and modify b's value by negation
  fmt.Println(b)
}
```

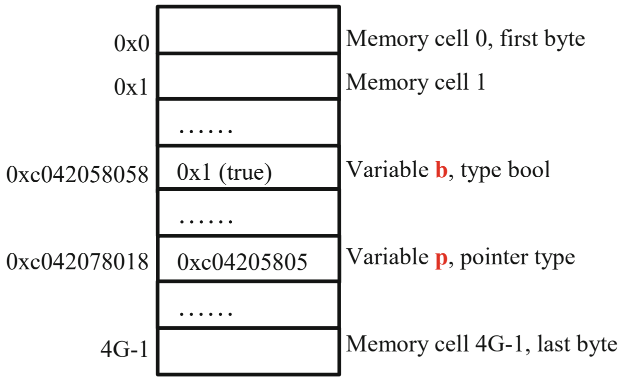
(a)

```
> go run ./pointer.go
0xc042058058
true
false
true
>
```

(b)



(c)



(d)

Fig. 5.9 Illustration of the pointer concept. (**a**) Source code pointer.go. (**b**) Screen display. (**c**) Diagram showing that p points to b. (**d**) Memory layout in a 4-GB byte addressable memory

For instance, p in an expression denotes the address of b, i.e., 0xc042058058. But *p denotes the value of b, i.e., the value at address 0xc042058058, which is initially true.

It is left as an exercise to print out the address of p. We can declare another pointer variable q to point to pointer variable p.

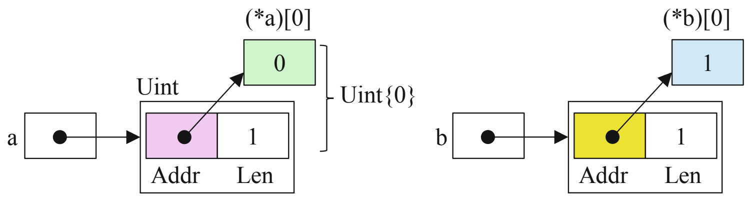
Example 5.11. Computing Fibonacci Numbers of Arbitrary Word Length
The fib.dp.big.go program introduced in Example 1.6 computes Fibonacci number F (n) for arbitrarily big integer n, using the dynamic programming method. A new integer data type big.Int from the math/big package is used to handle representation and arithmetic operations of big integers. However, the code is mostly provided by the Go libraries. It is powerful but has too many details for beginners to understand.

We use a simplified program fib.Uint.go to see how big integers are handled. This program is self-contained, and does not use the math/big package provided by the Go language. This fib.Uint.go program contains the definition of a new data type Uint, which is a slice of 64-bit unsigned integers, to represent an unsigned integer of arbitrarily big word length. An accumulator function Acc is defined to do a = a + b. A String function is defined for converting an unsigned integer into a string of decimal digits, to be printed out by fmt.Printf.

We focus on the fibonacci function definition. After the first two statements:

```
a = &Uint{0}
b = &Uint{1}
```

the program creates in memory the following two structures for variables a and b.



The data type Uint is a slice structure with two fields. One field holds the length. The other field holds the address pointing to the array of 64-bit unsigned integers. The notation Uint{0} denotes a slice of length 1, i.e., the array has one element holding the initial value of 0. The statement a=&Uint{0} assigns the address of Uint {0} to a. That is, a is a pointer variable holding the address of Uint{0}. Thus, the one element of the array is denoted by (*a)[0].

Two pointer variables a and b represent two unsigned integers of arbitrary length, where each unsigned integer is implemented by a slice of unsigned integers.

Let us look at the loop in Fig. 5.10a. Assume i is 1. The Acc(a, b) function realizes an accumulative addition a = a + b = 0 + 1 = 1. After the Acc(a, b) statement is executed, the memory contents change to the following configuration.

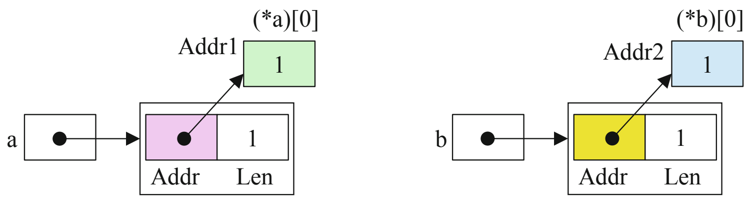
```
package main
import (
        "fmt"
        "math"
)
func main() {
        fmt.Printf("F(100) = %s\n", String(*(fibonacci(100))))
}
type Uint  []uint64
func fibonacci(n int) *Uint {
        a := &Uint{0}                          // a = 0
        b := &Uint{1}                          // b = 1
        for i := 1; i < n+1; i++ {
                Acc(a, b)                      //a = a + b
                a, b = b, a
        }
        return a
}
// Code defining Acc and String functions
```

(a)

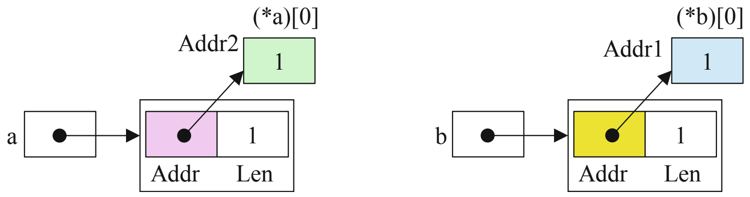
```
> go run fib.Uint.go
F(100) = 354224848179261915075
>
```

(b)

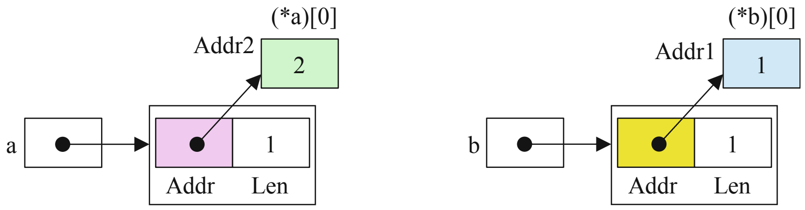
Fig. 5.10 Program fib.Uint.go and its output. (**a**) Source code of program fib.Uint.go. (**b**) Output by executing program fib.Uint.go



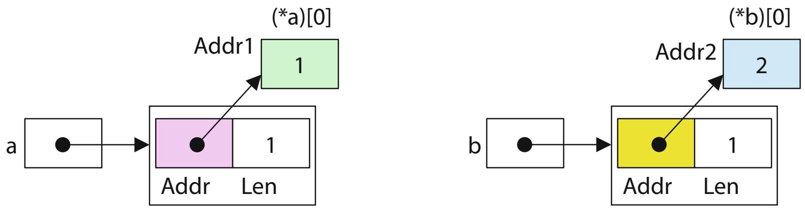
The a,b=b,a statement exchanges a and b to make sure that a is always the smaller of a and b. After the a,b=b,a statement is executed, the new memory configuration follows.



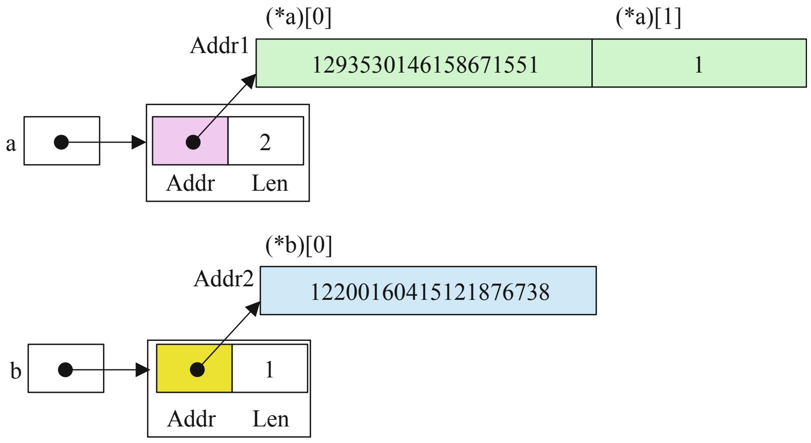
Assume i =2. After the Acc(a, b) statement is executed, the memory contents change to the following configuration.



After the a,b=b,a statement is executed, the new memory configuration follows.



Now assume i=93. This is the first time that the overflow occurs if we use a single 64-bit unsigned integer. After Acc(a, b) is executed, the memory becomes:



We avoid the overflow, which becomes a carry value into the second word, i.e., (*a)[1], of the slice of unsigned integers. Note that (*a)[1]=1 denotes 2^{64}. Change 100 to 93 in the program fib.Uint.go and see that it correctly outputs

F(93) = 12200160415121876738 and F(94) = 19740274219868223167.

☶

5.2.2.7 The File Abstraction

Files are used to organize and persistently store chunks of information. Here, **persistence** means files still exit when electrical power is turned off on a computer. A program **reads** a file from the hard disk into the memory for processing, and **stores** the modified results into the hard disk.

Files are stored in a **file system** of a computer. A file system organizes ordinary files and directories in a tree. A **directory** is a special file that contains other files. Each file, either an ordinary file or a directory, has a file name. Parts of the tree of file names of an example file system is shown in Fig. 5.11 and Table 5.4.

The **absolute file name** starts from the **root directory** "/" and goes down the tree, adding a slash "/" at each level. Each file has a unique absolute name. The file ucas. bmp has the absolute name /cs101/Prj2/ucas.bmp.

Let us look at Fig. 5.11 and Table 5.4 more carefully. When a user logs into a computer, she/he is automatically at a system-specified default directory, called the **home directory**. The directory in which the user is currently working is called the **current directory** or *working directory*. A pwd (Print Working Directory) command is used to print out the current directory.

Assume the user is at the home directory /cs101/, but wants to work in the working directory /cs101/Prj2/, which contains files for the second project. The

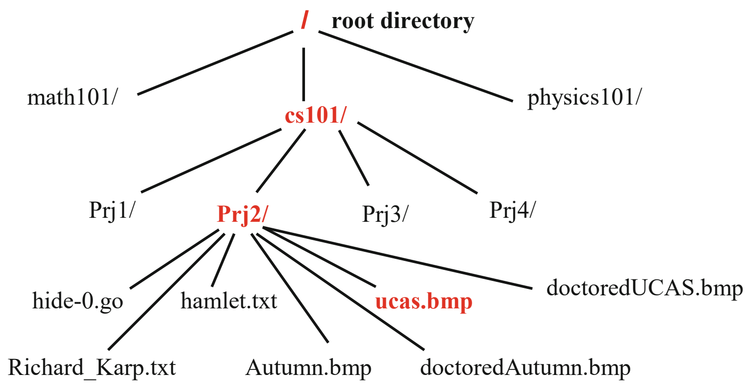


Fig. 5.11 An example tree of files and directories

Table 5.4 Typical directories and files

Typical directory and files	Absolute path name	Relative file name
Root directory	/	
Home directory	/cs101/	
Current directory		./
Parent directory		../
Program file to hide text in image	/cs101/Prj2/hide-0.go	hide-0.go
A text file	/cs101/Prj2/Richard_Karp.txt	Richard_Karp.txt
Another text file	/cs101/Prj2/hamlet.txt	hamlet.txt
An image file	/cs101/Prj2/ucas.bmp	ucas.bmp
A doctored image file	/cs101/Prj2/doctoredUCAS.bmp	doctoredUCAS.bmp
Another image file	/cs101/Prj2/Autumn.bmp	Autumn.bmp
Another doctored image file	/cs101/Prj2/doctoredAutumn.bmp	doctoredAutumn.bmp

user can execute the change directory command "cd Prj2" to change the current directory to /cs101/Prj2/. The sequence of screen printouts follows.

```
The user logs into a computer
>pwd    //the current directory is the home directory
/cs101/
>cd Prj2    //change to directory /cs101/Prj2
>pwd
/cs101/Prj2
```

Once in directory /cs101/Prj2/, the user can access all the seven files there using their shorter **relative file names**. When the current directory is /cs101/Prj2/, the following three names identify the same image file, and all three commands display the same image. A file name is also called a **path** or a path name.

```
>display Autumn.bmp
>display ./Autumn.bmp
>display /cs101/Prj2/Autumn.bmp
```

A file contains data and **metadata**. Data is the bits for the actual information provided by the file. Metadata is data about data, which provides additional information, such as the format and organization of data, the file name, the file size, the access permissions, the time of creation, etc. For instance, Autumn.bmp is a file containing 9144630 bytes and organized as shown in Fig. 5.12, where Pixel Array contains data, while BMP File Header and BMP Info Header contain metadata.

Such a **BMP** (bitmap) image file stores an image as an array of pixels. A **pixel** (picture element) represents a point of an image by three color depth values for the primary colors of red, green, and blue. Each color depth value of the RGB colors is represented as a number of uint8 byte type. Thus, each pixel needs three bytes.

A BMP image file's metadata includes information on the starting places and the sizes of various parts of the image, e.g., the width and the height of the image. Note

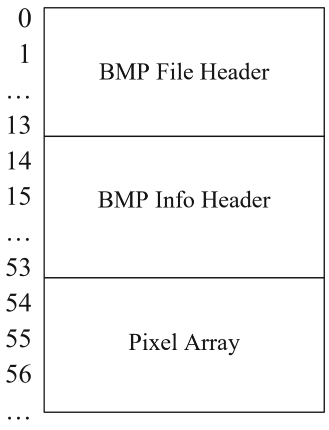


Fig. 5.12 Organization of data and metadata in a BMP image file

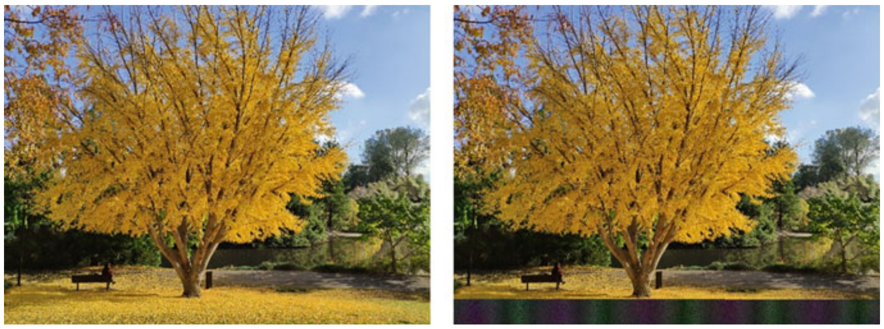


Fig. 5.13 Failure to hide text in a picture: the doctored image (the right picture) obviously differs from the original image (the left picture). (Photo credit: Chundian Li)

that the first 54 byte-level addresses are used for metadata. The pixel array for actual image data starts at address 54. The first pixel uses addresses 54, 55, and 56 to store its three RGB color depth values.

The black color is represented when the RGB values are set to (0, 0, 0), that is, when color depth values are all zero. Similarly, the white color is represented when the RGB values are set to (255,255,255), and the red color is represented when the RGB values are set to (255, 0, 0).

Example 5.12. Hide Text in a Picture by Doctoring the Image File
Let us appreciate the file abstraction in more details by hiding hamlet.txt in Autumn.bmp. That is, we doctor Autumn.bmp such that the content of the text file hamlet.txt are hidden in the picture of the image file Autumn.bmp.

- A careless hiding program will result in a doctored image that is visibly different from the original image, as shown in Fig. 5.13.
- A carefully designed hiding program will result in a doctored image that looks the same as the original image, as is shown in Fig. 5.14.

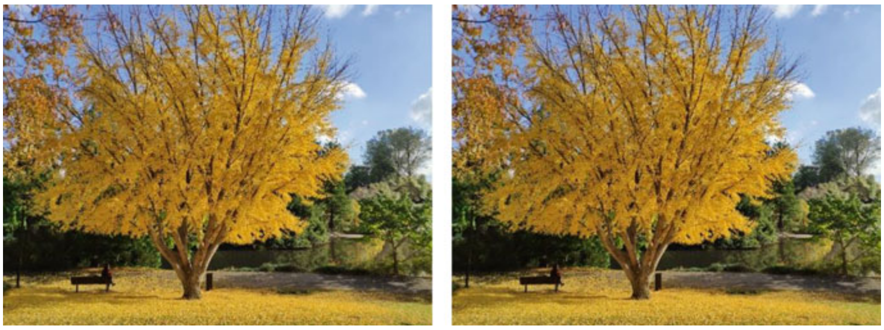


Fig. 5.14 Successfully hiding text in a picture: the doctored image (the right picture) shows no difference from the original image (the left picture). (Photo credit: Chundian Li)

This information hiding process is realized by the following algorithm, assuming the bmp file format of Fig. 5.12. The corresponding Go program is hide-0.go.

- **Input**: A text file hamlet.txt and an image file Autumn.bmp.
- **Output**: A doctored image file doctoredAutumn.bmp
- **Steps**:

1. Read Autumn.bmp into variable p // p for picture
2. Read hamlet.txt into variable t // t for text
3. Hide the length of hamlet.txt in the first 32 bytes of the Pixel Array
4. Hide hamlet.txt in variable p in the remaining bytes of the Pixel Array
5. Write p to file doctoredAutumn.bmp

We now discuss how to develop a program hide-0.go to realize this algorithm. All information is hidden in the Pixel Array, not in the metadata area. Every byte of information is hidden in 4 consecutive bytes of Pixel Array, such that only the *least significant two bits* of each byte of Pixel Array are modified.

Note that variable p is a slice of bytes. Pixel Array starts at address 54. That is, p [54:] holds the Pixel Array, and p[0:54] holds the metadata information.

To read the image file Autumn.bmp into variable p, the program executes

```
p, _ = ioutil.ReadFile("./Autumn.bmp").
```

The function ioutil.ReadFile is provided by Golang in the package “io/ioutil”. It accepts a text string of file name and returns the file contents as a byte slice. A Golang function call can return multiple values. If any value is no concern to us, we use an underscore “_” symbol as a placeholder.

Table 5.5 File access permissions (r: read; w: write; x: execute)

Owner			Group			Others		
r	w	x	r	w	x	r	w	x
1	1	0	1	1	0	1	1	0

Similarly, the contents of the text file hamlet.txt are read into a byte slice variable t by executing the following statement.

```
t, _ := ioutil.ReadFile("./hamlet.txt")
```

Modified contents of variable p are written to file doctoredAutumn.bmp by executing the following statement

```
ioutil.WriteFile("./doctoredAutumn.bmp", p, 0666)
```

where 0666 specifies the access permissions for file doctoredAutumn.bmp, according to the format of Table 5.5. The leading 0 bit indicates that this file is an ordinary file. A directory file should have a leading 1 bit.

The above WriteFile function call says that variable p is written to file doctoredAutumn.bmp. Being a new file, doctoredAutumn.bmp needs to be created first. The access permissions 666=110110110 says that the file's owner, the group which the owner belongs to, and other users have the rights to read and write the file. But, no user can access the file for execution. Using the shell command "ls -l", we can see the read-write-execute permissions listed as follows:

```
>ls -l doctoredAutumn.bmp
-rw-rw-rw-   ...  doctoredAutumn.bmp
>
```

Now let us go through the process of modifying variable p. This is done by repetitively calling a user-defined function modify(txt int, pix []byte, size int). The function saves an integer value *txt* into a byte slice variable *pix*, 2 bits at a time, for a total of *size* iterations.

```
func modify(txt int, pix []byte, size int) {
   for i := 0; i < size; i++ {
      replace last 2 bits of pix[i] with the last 2 bits of txt
      // the next iteration repeats with the next 2 bits of txt
   }
}
```

For instance, to hide character 'H' in p[86:90], we call modify(72, p[86:90], 4), and we have: txt is 'H' = 72 = 01001000; pix is p[86:90]; size is 4. The loop body is executed 4 times, and p[86:90] is modified as follows.

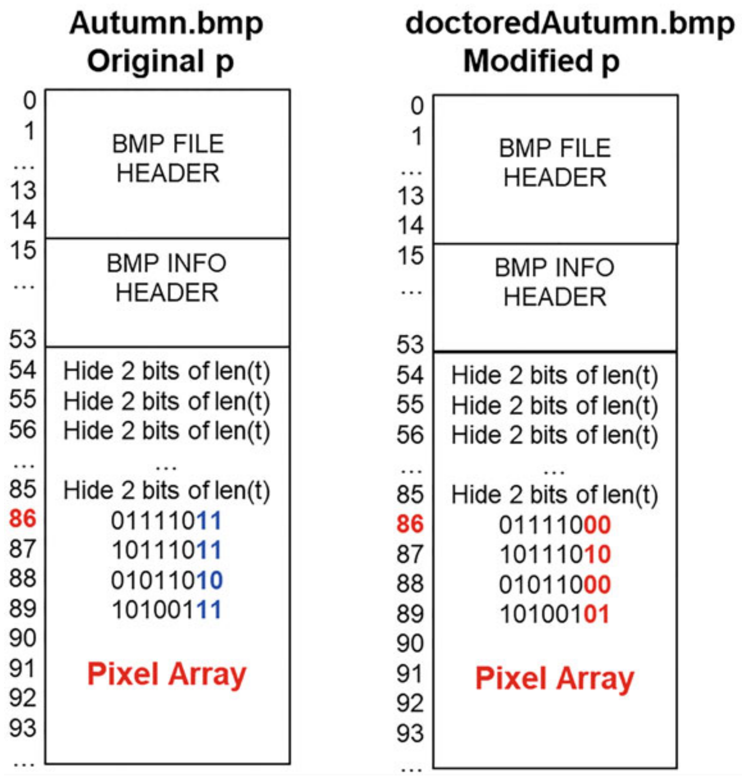
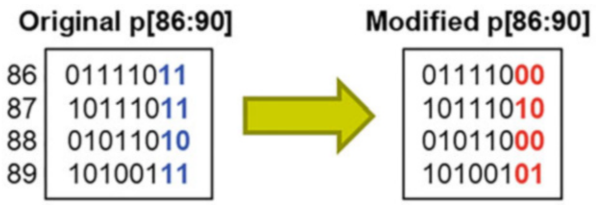


Fig. 5.15 Illustration of how the length and the first character 'H' of hamlet.txt are hidden



To hide hamlet.txt in Autumn.bmp and generate doctoredAutumn.bmp, we need to first save the length of the text file hamlet.txt, and then save the contents of hamlet.txt. This is illustrated in Fig. 5.15. The length needs to be saved in case we want to recover the text file from doctoredAutumn.bmp. This is done by executing

```
modify(len(t), p[S:S+T], T)
```

where S is the starting address of the pixel array, 54; and T is the length of the text file to be hidden. We assume the length len(t) is a 64-bit integer. Each byte of pixel array can hide 2 bits. Thus, to hide len(t) we need 64/2=32 bytes of Pixel Array. In other words, when T=32, slice p[54:86] is used for hiding len(t).

To hide the contents of hamlet.txt, we execute the following loop:

```
for i:=0; i<len(t); i++{
  offset := S+T+(i*4)
  modify(int(t[i]), p[offset:offset+C], C)
}
```

where each iteration of the loop body hides one character in four bytes of p at proper addresses. The first character of hamlet.txt is t[0]='H', which is hidden in p[86:90]. The second character is t[1]='A', hidden in p[90:94]. The third character is t[2]='M', hidden in p[94:98].

5.2.3 Control Abstractions

Five control abstractions are common in a high-level language program. Most of them are intuitive. We only elaborate loop and function.

- **Precedence** in an expression. For instance, in expression x*b+c || i < 7, the precedence ordering is ((x*b)+c) || (i < 7). When in doubt, use parentheses.
- **Sequencing**. By default, a sequence of statements is executed by the syntactic ordering of the sequence, one statement after another.
- **Selection**. An if-then-else statement, also called **conditional**. An example is

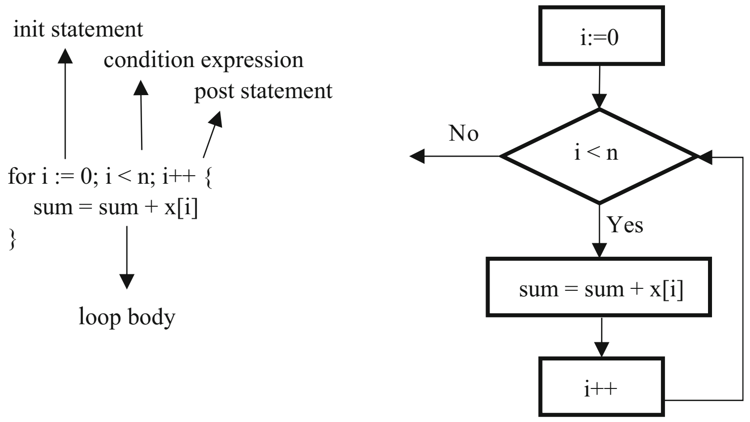
```
if i<7 {
  fmt.Println(i)
}
```

- **Loop** iteration. A loop repetitively executes a body of code. Each repetition is called an iteration. The body of code is called the loop body.
- **Function**. A function is defined once and can be called many times.

The following **for loop** statement sums the elements of an array.

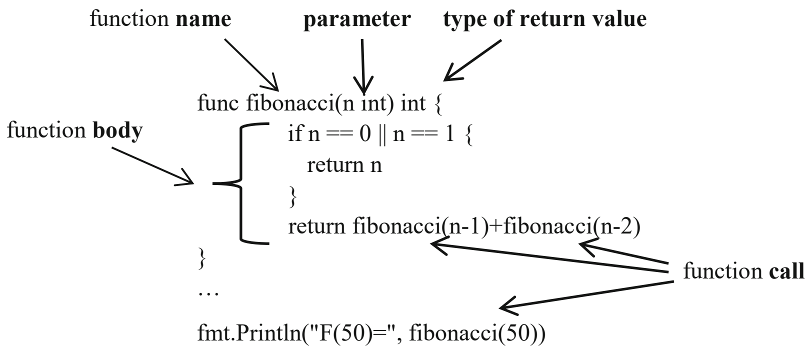
```
for i := 0; i < n; i++ {
    sum = sum + x[i]
 }
```

This loop statement has four parts, as shown in Fig. 5.16. The init statement i=0 sets the initial value of index i. The loop then checks the condition expression i < n. If false, the loop finishes. If true, executes the loop body, and then execute the post statement i++. After an iteration, the loop repeats by checking the condition again.



```
for i := 0; i < n; i++ {
   sum = sum + x[i]
}
```

Fig. 5.16 Illustration of a loop statement: its four parts and its control flow



```
func fibonacci(n int) int {
    if n == 0 || n == 1 {
      return n
    }
    return fibonacci(n-1)+fibonacci(n-2)
}
…
fmt.Println("F(50)=", fibonacci(50))
```

Fig. 5.17 A function is defined once and called three times in the above code

A **function** is a sub-program to be *called* by other statements in a program. A function definition starts with the keyword func and has four parts, as shown in Fig. 5.17: (1) a function *name* fibonacci, (2) an input *parameter* n of type int, (3) the type int of the function's *return value*, and (4) a function *body* enclosed between { and }.

This fibonacci function call appears three times in Fig. 5.17.

When a statement in a program calls a function, the execution environment (called **context**) is first saved before the program executes the function body, so that when the function returns, the program can resume execution with a proper context.

The operating system of a computer divides the memory space of the program (a process) into four segments, called **text**, **data**, **stack**, and **heap**, to hold program's code, static data variables and constants, function calls, and dynamic data, respectively. The context of a function call is saved in the stack segment (Fig. 5.18).

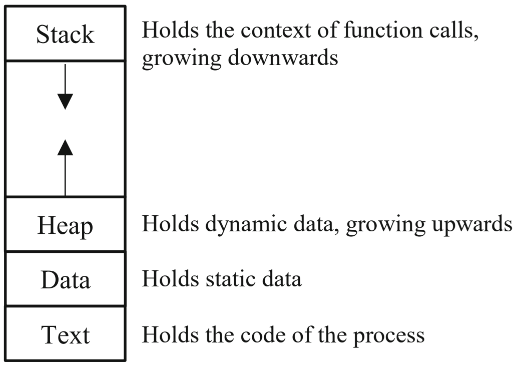

Stack
Holds the context of function calls, growing downwards
Heap
Holds dynamic data, growing upwards
Data
Holds static data
Text
Holds the code of the process

Fig. 5.18 Memory layout of the Text, Data, Stack, and Heap segments of a program (process)

5.3 Modularization

The divide-and-conquer methodology is discussed in algorithmic thinking. There is a similar methodology in systems thinking, called modularization. It has two facets: (1) dividing a system into multiple modules, and (2) composing modules into a higher-level abstraction, also known as system. When discussing modularization, the following points are noteworthy:

- Two modules may be interconnected, but they normally do not overlap.
- Modularization is a special form of abstraction where the information hiding principle is followed.
- Modularization, i.e., how to divide and compose a system, is an art, needing human imagination and creativity.

This section uses a number of progressively more complex examples to demonstrate modularization. Most examples are from the computer hardware.

5.3.1 Combinational Circuits

Combinational circuits are logic circuits using **gates** to realize propositional logic expressions. Figure 5.19 shows four gates realizing the four basic Boolean operators: AND, OR, NOT, and XOR.

Other gates can be realized by such basic gates. In reality, some basic gates are directly implemented by semiconductor circuitry. Figure 5.20 shows how a 2-input NAND gate is realized by a 4-transistor CMOS circuitry, where CMOS stands for Complementary Metal Oxide Semiconductor.

When X and Y are both at HIGH voltage level (logic 1), the lower two transistors are both ON and the upper two complementary transistors are both OFF. The output Z is connected to the ground (Vss) at LOW voltage level (logic 0). For any other

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

X	Z
0	1
1	0

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

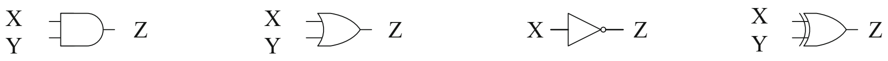
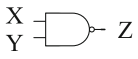


Fig. 5.19 Symbols of four basic gates AND, OR, NOT, and XOR, and their truth tables

X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0



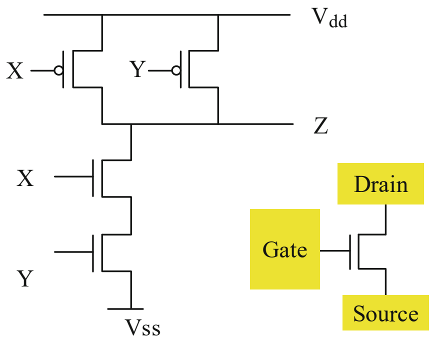


Fig. 5.20 NAND gate: truth table, symbol, and a CMOS implementation. Also shown are the three terminals of a transistor: source, drain, and gate

configuration, the output Z is connected to Vdd and thus at HIGH voltage level (logic 1). The CMOS semiconductor circuitry realizes a NAND gate.

The NAND gate is an abstraction with the COG properties:

- It is **c**onstrained by focusing on the Boolean logic functionality, ignoring details such as the voltage levels, number of transistors, power consumption, etc.
- It is **o**bjective. Its functionality is precisely defined by its truth table.
- It is **g**eneralizable. Its functionality can be implemented by a circuit other than the 4-transistor CMOS circuit.

In addition, the NAND gate symbol is much simpler than the 4-transistor CMOS circuit. It embodies the **information-hiding** principle: a module only exposes its interface and visible behaviors, but hides internal details and internal behavior. This is further illustrated in Fig. 5.21, which shows three equivalent representations of a combinational circuit: a Boolean expression, a logic circuit, and a CMOS circuit. The Boolean expression and the logic circuit hide the details of the CMOS circuit, and are much simpler.

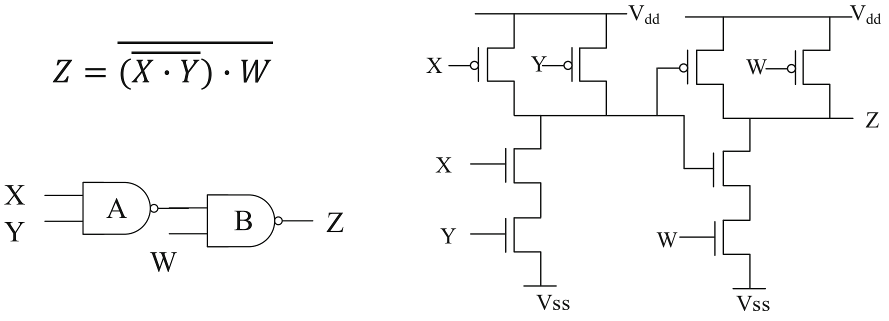
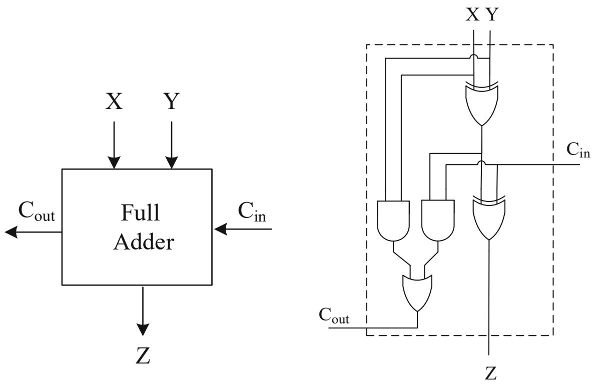


Fig. 5.21 A combinational circuit: Boolean expression, logic diagram, and CMOS circuit diagram

Fig. 5.22 Full adder: symbol and logic circuit diagram



5.3.1.1 Various Adders

In this UKA unit, students are shown several examples to see how to compose logic gates into various systems to do addition. Then as an exercise, students are asked to design a subtractor. Assume X and Y are unsigned integers, we want to compute Z = X + Y using gates.

Example 5.13. Full Adder and Ripple-Carry Adder

A very simple adder is a 1-bit adder, called **full adder**. The "full" here means that it considers also the carry-in and the carry-out bits, in addition to the two addend bits X, Y and the resulting sum bit Z. Figure 5.22 shows the full-adder symbol and its implementing logic circuit. Students are asked to verify that the correct Boolean expressions for the two outputs are $Z = X \bigoplus Y \bigoplus C_{in}$ and $C_{out} = (X \cdot Y) + (X \bigoplus Y) \cdot C_{in}$, respectively.

We can form an n-bit adder by cascading n full adders, as shown below for n=4 (Fig. 5.23).

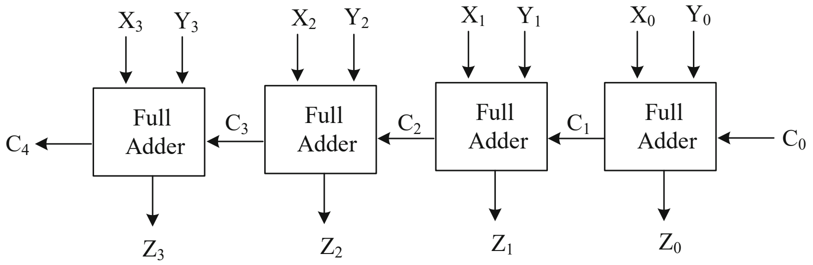


Fig. 5.23 A ripple-carry adder by cascading 4 full adders to form a 4-bit adder

Example 5.14. A Faster Adder
A ripple-carry adder serially generates the carry bits and the sum bits. A more efficient adder generates the carry and the sum bits in parallel. Such a 4-bit parallel adder is shown below, which computes X+Y=1011+1001 = 10100, with an overflowing carry bit of C_4=1. The trick is to compute all carry bits in parallel. The overflowing carry bit C_4 does not depend on lower carry bits any more (Fig. 5.24).

Example 5.15. An Adder-Subtractor Controlled by Multiplexers
The arithmetic-logic unit (ALU) of a processor supports many operations, not just the addition. How does it do that? By using control circuitry to select which operation to perform. The simplest control circuit is the 2-to-1 **multiplexer** in Fig. 5.25. The trapezoid is the multiplexer symbol, which selects one of the two input values X and Y as the output value Z, based on the selection value S. In other words, the multiplexer implements $Z = S \cdot Y + \bar{S} \cdot X$.

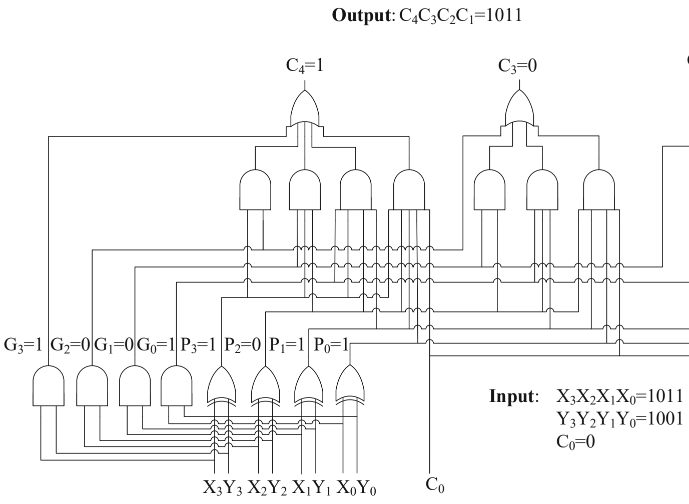
Figure 5.26 shows a 4-bit adder-subtractor designed using four multiplexers. The circuit is easy to understand by noting that subtracting Y from X, i.e., X − Y, is equivalent to adding the two's complement of Y to X, i.e., X + (−Y). For instance,

5-5=5+(-5) $\rightarrow$ 0101 + ($\bar{0}\,\bar{1}\,\bar{0}\,\bar{1}$ + 0001) =0101+1011=10000.

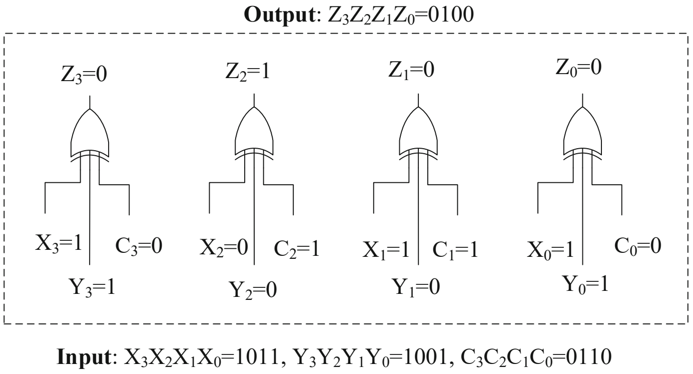
5.3.2 *Sequential Circuits*

Compared to combinational circuits, sequential circuits have one more type of components: state circuits. Thus, **sequential circuit = combinational circuit + state circuit**. With states, a system can execute multi-step computational processes. Each step computes two types of values: the current output values and the next state values. Both are computed from the current input values and the current state values.

States in hardware circuits are implemented by two types of basic circuits: (1) memory cells, and (2) **flip-flops**, also known as latches, which are logic circuits with feedback wires. Many sequential circuits use flip-flops to hold state values.



(a)



(b)

Fig. 5.24 A faster adder with parallel computing of carry bits. (**a**) Circuit to compute carry bits in parallel. (**b**) Circuit to compute sum bits in parallel

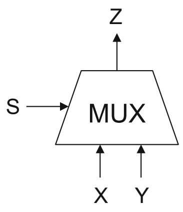
5.3.2.1 Various Types of Memory Cells

Four terms are often used for memory technology: DRAM, SRAM, NVM, ROM.

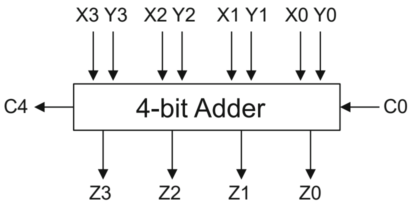
- Volatile memory means its contents are lost when power is turned off. Volatile memory is usually faster than non-volatile memory (NVM). There are two common types of volatile memory: DRAM and SRAM.
- Non-volatile memory means its contents are kept even when power is turned off. There are two common types of non-volatile memory: read-only memory (ROM) and read-write NVM.

Fig. 5.25 A 2-to-1 multiplexer: truth table, symbol, simplified truth table

S	X	Y	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



S	Z
0	X
1	Y



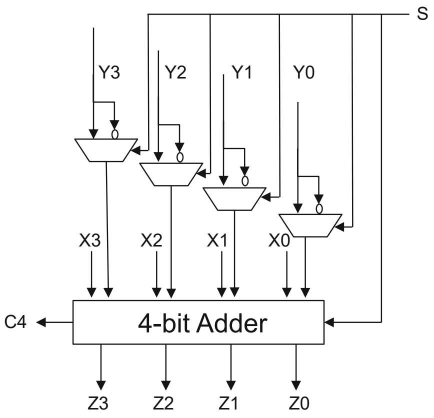


Fig. 5.26 A 4-bit adder and a 4-bit adder-subtractor, both in two's complement representation

A DRAM (dynamic random access memory) cell consists of a transistor and a capacitor and represents the state as the charge on the capacitor. This simplicity makes it inexpensive. However, capacitor leaks electricity. Thus, DRAM needs to constantly refresh its contents, once every 7.8–128 μs.

An SRAM (static random access memory) cell consists of six transistors. It is more expensive than DRAM but much faster. When the word line W is on, the bit line B is connected to the state Q to read or write. Suppose we want to write a 1 to the cell. Then B is set to 1, which causes Q to be HIGH (1) and the left-lower transistor to be on, turning $\bar{Q}$ to be LOW (0) (Fig. 5.27).

Read-only memory (ROM) often hardwires its contents into the memory hardware, thus does not lose its contents. The downside is that it cannot be written again. When the computer power is turned on, the computer usually fetches its first instruction from some ROM devices. Read-write NVM devices can be written again. A common example is the flash memory in a student's USB memory stick, also known as the USB flash drive or USB thumb drive.

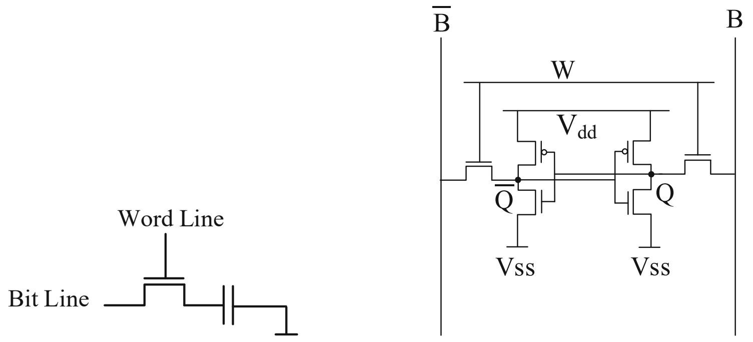


Fig. 5.27 A DRAM cell (left) and an SRAM cell (right) for storing one bit of state

5.3.2.2 A Logic Circuit with Feedbacks: The Delay Flip-Flop

In logic thinking, we try to avoid circular reasoning such as the barber paradox. But it turns out that adding feedback wires to a normal combinational circuit provides a new capability: now logic circuits can support states. Such circuits are called flip flops. We show below a **D flip-flop**, for *delay flip-flop*, which is widely used in computer circuits.

The D flip-flop is implemented with four NAND gates, with three feedback wires that are absent in normal combinational circuits. Its behavior is characterized by its truth table. When the Enable signal E is OFF (0), the D flip-flop maintains its state, that is, $Q_{next} = Q$. When the Enable signal E is ON (1), the D flip-flop changes its state to the D input value, that is, $Q_{next} = D$.

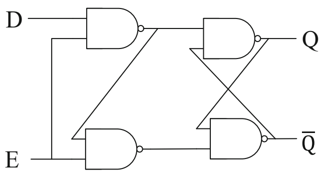
In practice, the system clock signal is often used for the Enable signal E. The clock signal, often written as CLK, is a special signal that alternates its value between LOW (0) and HIGH (1). Each cycle of 0 and 1 is called a **clock cycle**. The number of clock cycles in a second is called the **clock frequency**. Suppose a computer's processor (CPU) has a clock frequency of 3 GHz, i.e., 3 giga cycles per second. This translates to a clock cycle of 1/(3 GHz) = 0.33 ns.

Let us look at the behavior of a D flip-flop during a clock cycle, when E is replaced by CLK. When CLK=0, the D flip-flop maintains its state. When CLK=1, the D flip-flop changes its state to the value of D. Thus, after one clock cycle, the state Q of the D flip-flop changes its value to that of D. This behavior is why the flip-flop is called the *delay* flip-flop: its state output value delays one clock cycle from the input value D (Fig. 5.28).

5.3.2.3 A General Organization of Sequential Circuits

Any sequential circuit can be organized as shown in Fig. 5.29. It is a circuit comprised of three sub-circuits. The state circuit consists of one or more D flip-flops, where each flip-flop can hold one bit of state. The combinational circuit F

E	D	Q	Q_{next}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



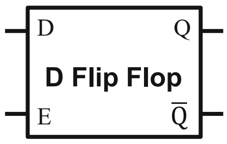


Fig. 5.28 The D flip-flop: its truth table, gates implementation, and symbol

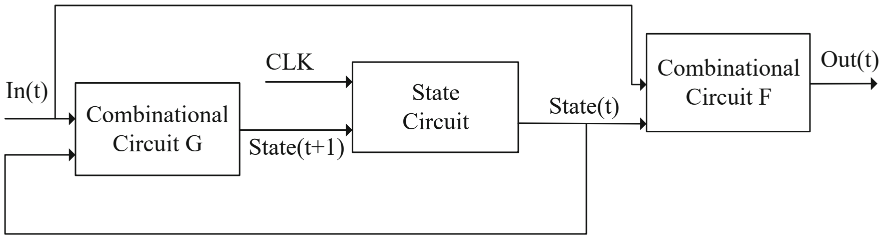


Fig. 5.29 A typical organization of sequential circuits

generates the current output value Out(t) from the current input value In(t) and the current state value State(t). The combinational circuit G generates the next state value State(t+1) from the current input value In(t) and the current state value State(t). In equation form, we have

- Out(t) = F(In(t), State(t))
- State(t+1) = G(In(t), State(t))

Designing a sequential circuit (let's call it the system) can follow this simple procedure: (1) find out the number *n* of bits needed for holding the system's states, and then use *n* D flip-flops to form the state circuit; (2) according to the system's requirements, design the combinational circuits F and G.

5.3.2.4 Serial Adder and Subtractor

This UKA unit asks students to go through the multiple steps of a serial addition process, to see how to design a sequential circuit and how a sequential circuit works. In each step, this 4-bit adder does a 1-bit full addition. Then as exercises, students are asked to design a 4-bit serial subtractor and an n-bit serial subtractor.

Example 5.16. A 4-Bit Serial Adder
Design a 4-bit serial adder of unsigned integers, which implements

Q	X	Y	Z	Q_{next}
q_0	0	0	0	q_0
q_0	0	1	1	q_0
q_0	1	0	1	q_0
q_0	1	1	0	q_1
q_1	0	0	1	q_0
q_1	0	1	0	q_1
q_1	1	0	0	q_1
q_1	1	1	1	q_1

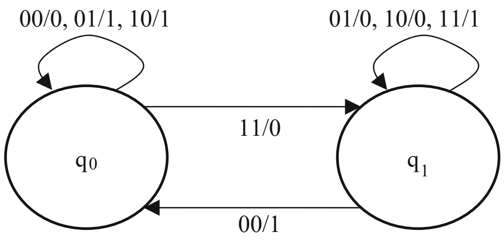


Fig. 5.30 Automaton for the 4-bit serial adder: state-transition table and state transition diagram

$$Z_3Z_2Z_1Z_0 = X_3X_2X_1X_0 + Y_3Y_2Y_1Y_0$$

in 4 steps, where each step does a 1-bit full addition. Verify the correctness of the design by executing the following addition

$11_{10} + 9_{10} = 1011_2 + 1001_2 = 10100_2 = 20_{10} = 4_{10}$ and overflow.

To design a sequential circuit, the first question to ask is: how many bits of the state the sequential circuit should have? It turns out that we can use the state to denote the current carry bit. Doing a binary addition serially, i.e., one bit at a time, we only have one carry bit to remember. Thus, we only need one D flip-flop to hold one bit of state, which can have two state values, q_0 and q_1, to denote the current carry value to be 0 and 1, respectively.

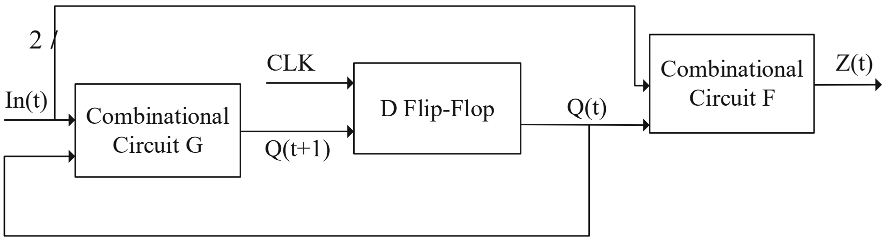
From the semantics of the binary addition of unsigned integers, we can derive a state-transition table and the equivalent state transition diagram of the automaton for the 4-bit serial adder, as shown in Fig. 5.30. Note that X, Y, Z, Q, and Q_{next} denote the current bits of the input X, the input Y, the output Z, the state, and the next state, respectively. The notation XY/Z attached to an arrow needs a couple of examples to explain. 01/1 says that in state q_0, if XY=01, then output Z=1 and stays in state q_0. 11/0 says that in state q_0, if XY=11, then output Z=0 and transition to state q_1.

From the general organization of sequential circuits, i.e., Fig. 5.29, we can obtain the first diagram shown in Fig. 5.31 for the serial adder. The enable signal of the D flip-flop is the clock signal CLK. The input In(t) now denotes two variables X and Y. The second diagram in Fig. 5.31 further simplifies by using notations closer to the state-transition table. From the transition table, we can easily derive the following Boolean expressions for the two combinational circuits F and G.

- Z(t) = F(In(t), Q(t))
- Q(t+1) = G(In(t), Q(t))

which are equivalently rewritten as the following expressions:

- Combinational circuit F: $Z = F(X, Y, Q) = X \bigoplus Y \bigoplus C$
- Combinational circuit G: $Q_{next} = G(X, Y, Q) = (X \cdot Y) + (X \bigoplus Y) \cdot Q$



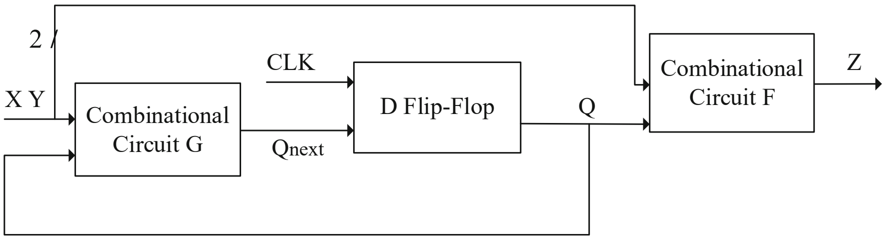


Fig. 5.31 A typical organization diagram of the serial adder

These are the current output function and the next state function, respectively. They denote the current output bit Z and the carry-out bit C_{out}, respectively. That is, $Q = C_{in}$ and $Q_{next} = C_{out}$.

Now we verify the correctness of the serial adder with an example:

$11_{10} + 9_{10} = 1011_2 + 1001_2 = 10100_2 = 20_{10} = 4_{10}$ and overflow.

Note that we have as inputs

$$X_3X_2X_1X_0 = 1011$$

$$Y_3Y_2Y_1Y_0 = 1001$$

and we want the correct output to be

$$Z_3Z_2Z_1Z_0 = 0100$$

with an overflowing carry-out bit of value 1.

This can be done in a sequence of four steps, where each step does a 1-bit full addition. We immediately found an error in the above design. It did not set an initial value for the state.

Setting the initial state value (the lowest carry-in bit C_0) to 0, we have:

- Step 1: $Z_0 = X_0 \oplus Y_0 \oplus C_0 = 1 \oplus 1 \oplus 0 = \mathbf{0}$;
  $C_1 = (X_0 \cdot Y_0) + (X_0 \oplus Y_0) \cdot C_0 = (1 \cdot 1) + (1 \oplus 1) \cdot 0 = 1$
- Step 2: $Z_1 = X_1 \oplus Y_1 \oplus C_1 = 1 \oplus 0 \oplus 1 = \mathbf{0}$;
  $C_2 = (X_1 \cdot Y_1) + (X_1 \oplus Y_1) \cdot C_1 = (1 \cdot 0) + (1 \oplus 0) \cdot 1 = 1$

- Step 3: $Z_2 = X_2 \bigoplus Y_2 \bigoplus C_2 = 0 \bigoplus 0 \bigoplus 1 = \mathbf{1}$;
  $C_3 = (X_2 \cdot Y_2) + (X_2 \bigoplus Y_2) \cdot C_2 = (0 \cdot 0) + (0 \bigoplus 0) \cdot 1 = 0$
- Step 4: $Z_3 = X_3 \bigoplus Y_3 \bigoplus C_3 = 1 \bigoplus 1 \bigoplus 0 = \mathbf{0}$;
  $C_4 = (X_3 \cdot Y_3) + (X_3 \bigoplus Y_3) \cdot C_3 = (1 \cdot 1) + (1 \bigoplus 1) \cdot 0 = \mathbf{1}$

The final output is $Z_3Z_2Z_1Z_0 = \mathbf{0100}$, with the carry $-$ out $C_4 = \mathbf{1}$, indicating that an overflow has occurred. The carry-out and the sum output bits are written as

$$C_4Z_3Z_2Z_1Z_0 = 10100.$$

The computational result is correct.

5.3.3 Instruction Set and Instruction Pipeline

The sequential circuits discussed in the last section are actually **synchronous sequential circuits**, because each circuit is driven by a common clock signal CLK. All states of the circuit transition to their respective next states at each clock cycle. Such synchronous sequential circuits behave the same as the automata discussed in Sect. 3.2. The former implements the latter.

Automata and sequential circuits are basic concepts and widely used in computer systems and computer application systems. This section discusses how they are used in the processor (CPU) of a computer, to implement the instruction pipeline. Each stage of the instruction pipeline is implemented as a sequential circuit.

Every processor has an **instruction set**, which is the set of all possible instructions of the processor, organized in a systematic way. The **instruction pipeline** is the hardware that executes the instructions. A 3-stage instruction pipeline is shown in

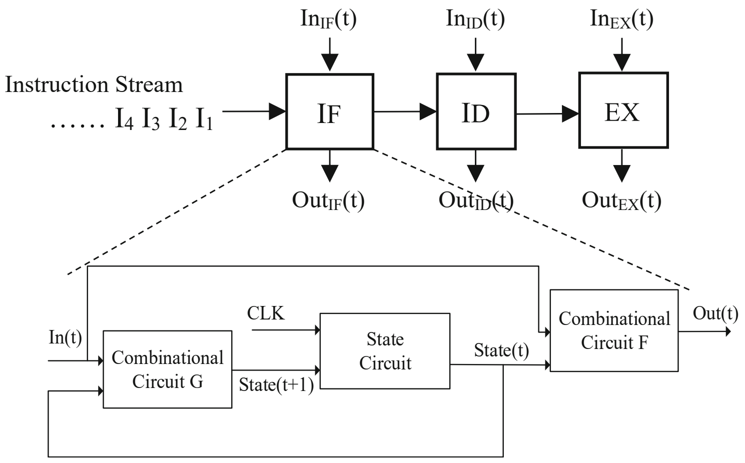


Fig. 5.32 A 3-stage instruction pipeline, each stage implemented by a sequential circuit

Fig. 5.32. The three stages are instruction fetch (IF), instruction decode (ID), and instruction execute (EX) stages.

An instruction is first fetched from the memory to an *instruction register* (IR) in the processor. It is then decoded to generate proper control signals. The control signals are then applied, together with the clock signal, to drive the multiplexers, thus to execute the instruction and produce the result. For instance, the processing of the MOV 0, R1 instruction stores an immediate value 0 to register R1. It goes through the following three stages.

- Instruction Fetch (IF): IR ← M[PC]
- Instruction Decode (ID): Signals = Decode(IR)
- Instruction Execute (EX): R1 ← 0; PC ← PC+1

Having more pipeline stages can help increase the clock frequency, thus making the processor faster. Modern processors have 5–31 pipeline stages.

Example 5.17. Design a Simple Instruction Set
Recall Sect. 2.3, where we used an assembly language program to realize the following Go loop structure:

```
for i := 2; i < 51; i++ {
  fib[i] = fib[i-1] + fib[i-2]
}
```

The code snippets of the Go language program and the corresponding assembly language code are shown below side by side, to show their correspondences.

```
fib[0] = 0                                 MOV 0, R1
                                           MOV R1, M[R0]    //R0=12 initially
fib[1] = 1                                 MOV 1, R1
                                           MOV R1, M[R0+8]
for i := 2; i < 51; i++ {                  MOV 2, R2   // i:=2
 fib[i] = fib[i-1] + fib[i-2]      Loop:   MOV 0, R1   // label Loop
                                           ADD M[R0+R2*8-16], R1
                                           ADD M[R0+R2*8-8], R1
                                           MOV R1, M[R0+R2*8-0]
                                           INC R2     // i++
                                           CMP 51, R2   // i < 51?
}                                          JL Loop     // if Yes, goto Loop
```

Suppose this is all we want this **Fibonacci computer** to do. That is, it only needs to execute the above 11 instructions. Design an instruction set for this computer.

An instruction consists of *opcode* and *operands*. The design process can follow the following procedure: (1) Determine the types of instructions and decide the opcodes; (2) for each opcode, determine its operands. We may need to do tradeoffs to balance the design of the entire instruction set.

This **Fibonacci computer** has five registers visible to the user: FLAGS, PC, R0, R1, and R2. There are six different types of instructions, and each is assigned an

Table 5.6 The opcodes of the instruction set of the Fibonacci computer

Instruction type	Opcode	Semantics
MOV to Register	000	Assign an immediate value to a register
MOV to Memory	001	Assign the content of a register to M[Address]
ADD	010	R1 + M[Address] → R1
INC	011	R + 1 → R (R is a register)
CMP	100	Compare to a value, assign the result to FLAGS
JL	101	If FLAGS is '<' (less than), Loop → PC

Table 5.7 How the 11 instructions of the Fibonacci computer are represented

Opcode 3-bit	Operand 1 Immediate Value, 6-bit	Operand 2 Register, 2-bit	Instruction
000	000000	01	MOV 0, R1
000	000001	01	MOV 1, R1
000	000010	10	MOV 2, R2
011	111111	10	INC R2
100	110011	10	CMP 51, R2
101	00000101		JL Loop
Opcode 3-bit	Operand 1 Memory Address, 6-bit	Operand 2 Register, 2-bit	Instruction
001	R0+R2*0+0	01	MOV R1, M[R0]
001	R0+R2*0+8	01	MOV R1, M[R0+8]
001	R0+R2*8-0	01	MOV R1, M[R0+R2*8-0]
010	R0+R2*8-8	01	ADD M[R0+R2*8-8], R1
010	R0+R2*8-16	01	ADD M[R0+R2*8-16], R1

opcode, as is shown in Table 5.6. Note that the three instructions (1) MOV 0, R1; (2) MOV 1, R1 and (3) MOV 2, R2 belong to one type of instruction: it moves an immediate value to a register. We only need three bits to hold the six opcodes.

The instructions, each needing 11 bits to represent, are organized in two groups as shown in Table 5.7. The JL Loop (jump to Loop if FLAGS is "less than") instruction needs only one operand to hold the value of Loop, which in this example is 5. The other instructions each need two operands.

Let us first look at the first group of instructions, where the first operand is an immediate value. The JL instruction (opcode 101) needs only 1 operand, which is an unsigned integer of 8 bits ($11 - 3 = 8$). The JL instruction can jump to a memory address from 0 to 255. For the other 5 instructions, Operand 1 is a 6-bit value, and Operand 2 specifies one of four registers. That is, 00, 01, 10, 11 specify R0, R1, R2, R3, respectively.

In the second group of instructions, Operand 1 is a 6-bit value specifying a memory address, and Operand 2 specifies one of four registers. Based on the base +index+offset addressing mode, the memory address is computed by the following relations:

```
Address = R0 + R2*I + J, where
  I = 0, 1, 2, 4, 8
  J = 0, ±4, ±8, ±16
```

Since the base register R0 and the index register R2 are given and fixed, there are $5 \times 7 = 35$ possible (I, J) pairs, and thus 35 distinct values for Operand 1. Since $35<2^6$, 6 bits are enough for Operand 1. For instance, given the initial values of R0 = 12 and R2 =2, R0+R2*8-8 = 12 + 2*8 -8 = 12 + 8 = 20.

5.3.4 Software Stack on a von Neumann Computer

Now a computer can execute instructions by its instruction pipeline hardware, we go to see how the computer software is organized. Typically, the software is organized as a layered structure, called **software stack**, on top of a common abstraction of hardware, the von Neumann architecture, as shown in Table 5.8.

Software can be classified as **application software** and **infrastructure software**. The latter provides an infrastructure for applications and can be further divided into **middleware** and **system software**. Middleware is so called because it is between application software and system software. Examples of middleware include database management systems such as MySQL, Web servers such as Nginx, and Web Browsers such as Chrome. System software normally views middleware the same as application software.

We have already used the Linux operating system and the Go programming language with its associated Go compiler. In the Personal Artifact project, students

Table 5.8 Examples of software stack on top of a common von Neumann architecture

Software type			Example
Application Software			Scientific computing, Business computing, Personal productivity software;PDF, Search Engine, TikTok, WeChat
Infrastructure Software	Middleware	Databases Web servers Web Browsers	MySQL Nginx, WebServer.go Chrome, Safari
	System Software	Languages Compilers Interpreters	C, Go, JavaScript, Python Shell
		Operating Systems	Linux, Android, iOS, Windows
		Firmware	BIOS
von Neumann Architecture			
Hardware			

are asked to use the HTML/CSS/JavaScript programming environment to create their dynamic webpages, to be run on a Web server and a Web browser.

The most basic system software is called **firmware**, often hardwired into a memory device. This way, when the power is turned on, the instructions and data in the firmware are ready to go. Thus, firmware is not as hard as hardware, but also not as flexible or soft as software. Firmware is used to realize functionality such as power-on checks and diagnostics, initializing the basic hardware configuration, and loading the operating system from the hard disk. An example firmware is **BIOS**, or the Basic Input/Output System in our personal computers.

Source code refers to programs written in human understandable high-level languages or assembly languages. *Binary code* (executable code) refers to the code of a string of 0's and 1's.

5.4 Seamless Transition

A modern personal computer can execute a billion instructions per second. How does the computer smoothly transition from one instruction to the next one in this billion-step computational process? We can refine this question by asking three more focused problems:

- How to identify the first instruction?
- How to ensure a single instruction's correct execution?
- How to find the next instruction and transition to it?

Computer systems capable of solving the above three problems are said to have the capability of **seamless transition**. It turns out that computer science has established four principles to support seamless transition, as listed in Box 5.1.

Box 5.1. Four Principles of Seamless Transition

- **Yang's Cycle Principle**. A computational process consists of cycles of diversity. The system finishes one cycle and automatically returns to the beginning (of the next cycle), so that processes preserve their kinds.
- **Postel's Robustness Principle**. Be tolerant of inputs, and strict on outputs.
- **von Neumann's Exhaustiveness Principle**. Instructions must be given to the computer in absolutely exhaustive detail, for the computer to execute completely without intelligent human intervention.
- **Amdahl's Law**. Suppose a computational process's execution time can be broken into two portions $(1-f)$ and f, such that $(1-f)>f$. Improve on the common case, i. e. , the $(1-f)$ portion, but be aware that speedup is at most $1/f$.

It is not the case that one principle is proposed for a problem. The roles of the four principles can be roughly described as follows:

- Yang's cycle principle addresses part of the third problem as well as the mega question: how does the computer smoothly transition from one instruction to the next one?
- Postel's robustness principle mostly addresses the second problem.
- von Neumann's exhaustiveness principle involves all three problems.

While the above three principles target the functionality of a system, the fourth principle, Amdahl's law, addresses the performance of a system. Seamless transition does not just mean that the system correctly executes computational processes, it also implies that computational processes flow through the system smoothly and seamlessly. Amdahl's law addresses such performance issues.

5.4.1 *Yang's Cycle Principle*

Any computational process is a multi-step process. A fundamental question is the following: How to ensure the seamless transition from one step to the next step?

Computer science uses a single principle to solve this problem. This principle does not have a name. We call it Yang's cycle principle, or Yang Xiong's principle of cycles, because Yang Xiong presented a similar principle around year 2 BCE, in his classic *The Canon of Supreme Mystery* (太玄经).

《太玄经・周首》：𝌇 阳气周神而反乎始，物继其汇。

Head Zhou (Full Circle) 𝌇: Yang qi comes full circle. Divinely, it returns to the beginning. Things go on to preserve their kinds.

A system executes a computational process in a sequence of cycles. The system finishes one cycle and automatically returns to the beginning (of the next cycle), so that different computational processes preserve their respective kinds.

How can it be done to automatically return to the beginning of the next cycle? Because, at the basic level, a computing system is a sequential circuit. It uses the current state Q to generate the next state Q_{next}.

For instance, recall the typical organization of sequential circuit in Sect. 5.3.2, which we redraw below. At step *k*, the system is in state Q, which is the output of the D flip-flops. The system uses Q and the current input In to generate Q_{next} through circuit G, and to generate the current output Out through circuit F. This is the functionality of step *k*. When step *k* finishes, Q_{next} replaces Q to become the current state via the D flip-flops, and the system returns to the beginning of step $k + 1$.

The same mechanism works for all steps, enabling different steps to realize different functionalities. This is what "things go on to preserve their kinds" implies.

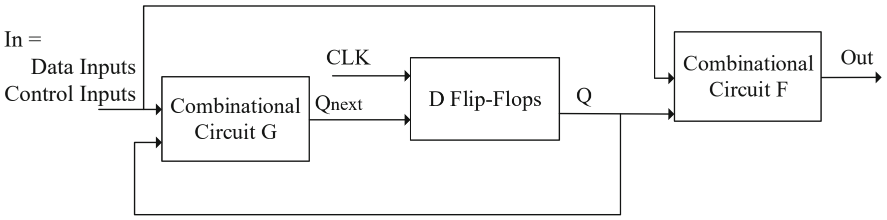


Fig. 5.33 A typical sequential circuit organization

This support of diversity can be realized by using control signals. In Fig. 5.33, inputs (In) to the two circuits F and G actually consist of Data Inputs and Control Inputs. A control input signal can be used to select the functionality of a circuit, such as to add or to subtract, as shown by the selection signal S in Fig. 5.26.

The above principle is a general principle, working for step (or cycle) that could be at different granularities:

- A small granularity is **clock cycle**. For instance, a 1-GHz processor has the clock cycle of 1 ns. At this granularity, the multi-step computational process is a sequence of clock cycles. The system finishes one clock cycle and returns to the beginning of the next clock cycle.
- At the instruction granularity, a computer finishes a step by executing an **instruction cycle**. At this granularity, the multi-step computational process is a sequence of instruction cycles. The system finishes one instruction cycle and returns to the beginning of the next instruction cycle. Note that an instruction cycle is itself implemented by a sequence of clock cycles, to realize pipeline stages of Instruction Fetch, Instruction Decode, and Instruction Execution. Each stage of the instruction pipeline may need one or more clock cycles.
- At the program granularity, a computer finishes a step by executing a **program cycle**. At this granularity, the multi-step computational process is a sequence of program cycles. The system finishes one program cycle and returns to the beginning of the next program cycle. Note that a program cycle is itself implemented by a sequence of instruction cycles.

To recap: a task such as sending a WeChat text message involves the execution of a number of programs. A task is implemented by a sequence of program cycles. A program cycle is implemented by a sequence of instruction cycles. An instruction cycle is implemented by a sequence of clock cycles. In each of these cases, we observe a common principle: when a step finishes, the system returns to the beginning of the next step, until the entire computational process completes.

5.4.2 *Postel's Robustness Principle*

Postel's robustness principle was originally proposed in 1980 by Internet pioneer Jon Postel for robust communication on the Internet. Since then, it has been applied to other systems.

TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.
Jon Postel, 1980

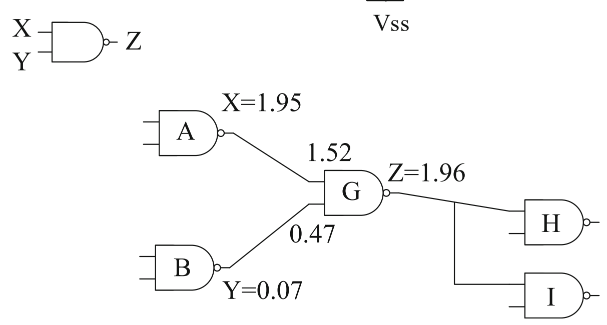
This principle is often shortened to: be tolerant of input and strict on output. It has an important implication: accumulation of errors, drifts, and distortions can often be avoided. We will use an example of combinational circuit design to illustrate this principle.

Figure 5.34 shows a combinational circuit of five NAND gates. We focus on gate G. It receives inputs from gates A and B, and sends an output to gates H and I. We need to understand how the single step of gate G correctly works. That is, how gate G correctly accepts values from gates A, B and generates correct output value Z.

Two designs of the CMOS circuit for a NAND gate are shown by contrasting their parameters. Suppose the CMOS transistors have the following characteristics:

- The HIGH voltage Vdd is 2 Volt.
- The LOW voltage Vss is 0 Volt.

X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0



Parameter Settings

Vdd = 2 Volt
Vss = 0 Volt
Vth = 0.7 Volt

Naïve Design
Logic 1: > 0.7 Volt
Logic 0: < 0.7 Volt

Better Design
For Input Voltages
Logic 1: > 1.5 Volt
Logic 0: < 0.5 Volt
For Output Voltages
Logic 1: > 1.9 Volt
Logic 0: < 0.1 Volt

Fig. 5.34 A combinational circuit demonstrating Postel's Robustness Principle

- The threshold voltage Vth is 0.7 Volt. That is, the transistor will turn ON if the gate voltage rises over the threshold of 0.7 Volt, and turn OFF when the gate voltage drops below 0.7 Volt.

The naïve design has various problems and is difficult or impossible to implement. The main problem is that the specification "Logic 1: > 0.7 Volt" and "Logic 0: < 0.7 Volt" has no safe margin for deviations and leaves no minimal gap between voltages for logic 1 and logic 0. How many margins should there be? Should inputs and outputs of a logic gate be treated the same way?

Postel's robustness principle tells us No! We should leave more margin for the input than the output. The Better Design follows this principle.

- *Tolerance on inputs*. The Better Design leaves a 0.5 Volt margin for an input signal to a gate. Only when the input voltage is larger than 1.5 Volt, not 0.7 Volt, should the logic value correspond to logic 1. Only when the input voltage is less than 0.5 Volt, not 0.7 Volt, should the logic value correspond to logic 0. There is a minimal gap of 1 Volt between the voltages for logic 0 and logic 1.
- *Strictness on outputs*. The Better Design leaves a 0.1 Volt margin for an output signal from a gate. Only when the out voltage is larger than 1.9 Volt, not 0.7 Volt, should the logic value correspond to logic 1. Only when the output voltage is less than 0.1 Volt, not 0.7 Volt, should the logic value correspond to logic 0. There is a minimal gap of 1.8 Volt between the voltages for logic 0 and logic 1.

With this design, a gate can tolerate input deviations within 0.5 Volt, but accommodate output deviations within only 0.1 Volt. The design is indeed tolerant of input and strict on output.

Suppose the two inputs of gate G have logic values X=1 and Y=0. The output Z should be equal to 1. But Gate B actually outputs 0.07 Volt. The signal is later raised to 0.47 Volt when it travels through the wire to arrive at gate G. This is fine, because gate G views any input value between 0 Volt and 0.5 Volt as indicating a logic 0. Similar analysis applies to the output of gate A. Gate G views any input value between 2 Volt and 1.5 Volt as indicating a logic 1. For Z=1, the output voltage is not between 2 Volt and 1.5 Volt. The CMOS circuit implementation guarantees that it is a valued between 2 Volt and 1.9 Volt.

5.4.3 von Neumann's Exhaustiveness Principle

The exhaustiveness principle is due to John von Neumann. In the *First Draft of a Report on the EDVAC*, he stated right at the beginning of the document the following principle, when defining an automatic computing system (called *the device*) for problem-solving such as to solve a non-linear partial differential equation (called *this operation*).

The instructions which govern this operation must be given to the device in absolutely exhaustive detail. They include all numerical information which is required to solve the problem under consideration . . .

Once these instructions are given to the device, it must be able to carry them out completely and without any need for further intelligent human intervention.

John von Neumann, 1945

Note that instructions in the above quote are not only binary instructions of program code, but *all numerical information*. Obviously, the computer must be given the input data and the processing program code. The computer must also be given information such as the library of functions, context information, etc.

In addition, a computer must be given the answers to the following three questions:

- Where and what is the first instruction, when the computer power is turned on?
- How to determine the next instruction to execute?
- What types of exceptions are there, to normal execution of programs?

We use three examples to demonstrate the exhaustive principle.

Example 5.18. First Instruction to Execute When the Power Is Turned On When the power is turned on in a computer with an x86 processor, the first instruction to execute is at memory address 0xFFFFFFF0, and it contains a jump instruction such as JUMP 000F0000. Address 000F0000 contains the entry instruction for the BIOS code.

Thus, the computer starts by executing the BIOS firmware code to initialize the system and to load the operating system. Adding a jump instruction upfront increases flexibility. For instance, if we want the computer to start by executing another firmware code BIOS-2 at address 000FA000, we can easily do so by changing the contents of address 0xFFFFFFF0 to JUMP 000FA000.

Example 5.19. Three Methods to Determine the Next Instruction to Execute Historically, three methods have been used to determine the next instruction to execute. Modern computers mostly use the program counter (PC) mechanism: the address of the next instruction to execute is stored in the program counter.

Another simple method was used by the revised version of the ENIAC computer: the format of every instruction is expanded to have a field for holding the address of the next instruction.

The earliest method is linear sequencing. The Harvard Mark I computer was designed by graduate student Howard Aiken and built by IBM. It was an electromechanical *Automatic Sequence Controlled Calculator*, meaning that instructions are linearly sequenced. There is no jump or branch. The next instruction is located right after the current instruction on an instruction tape.

The Mark I machine treats instructions and data differently, by storing data in memory and instructions on tape. This **Harvard architecture** is still widely used in the cache units of modern computers. A processor normally has separate instruction cache and data cache. In contrast, the **Princeton architecture** uses a single cache or memory to store both data and instructions.

Example 5.20. Three Types of Exceptions
Three types of **exceptions** are considered in almost all computers.

- *Interrupt*. For instance, when the user punches a key on the keyboard, the current instruction finishes and the computer jumps to an interrupt handling subprogram to handle the interrupt, such as writing the punched key value to memory.
- *Hardware error*. When the memory chip becomes faulty, we cannot use the interrupt mechanism, since we cannot fetch the current instruction from memory. An exception handling subprogram stored somewhere else should be executed.
- *Machine check*. This is the "all other" exception, for exhaustiveness.

5.4.4 (***) Amdahl's Law

Amdahl's law was originally proposed by Gene Amdahl, a designer of the famous IBM S/360 general-purpose computer. The modern form of this law can be stated succinctly as follows: After enhancing a portion of a system, the speedup obtained is upper bounded by the reciprocal of the other portion's time.

More precisely, suppose a system's execution time is broken into two portions $(1-f)$ and f, such that $(1-f) > f$. Enhancement on the $(1-f)$ portion can lead to a speedup no more than $1/f$.

Here, **speedup** is (time before enhancement) / (time after enhancement).

Example 5.21. How Much Speedup Is Possible by Enhancing the Processor?
Suppose executing a task, e.g., rendering a short movie, takes 600 s. CPU processing takes 90% of time, and accessing the memory and I/O devices takes the remaining10%. How much is the speedup, if the CPU is 9 times faster? How much is the speedup, if the CPU is 9999 times faster?

For the old system, $f = 0.1$, and the execution time of the task is

$$T_{old} = 0.9 \times 600 + 0.1 \times 600 = 540 + 60 = 600 \text{ s.}$$

For the enhanced system, the CPU is 9 times faster, implying that the CPU processing time is 1/10 of the old one. The execution time becomes

$$T_{enhanced} = 0.9 \times 600/10 + 0.1 \times 600 = 54 + 60 = 114 \text{ s.}$$

The speedup is $600/114 = 5.26$.

Now suppose the new CPU is 9999 times faster. In the enhanced system, the CPU processing time is 1/10000 of the old CPU processing time. The execution time of the task becomes

$$T_{enhanced} = \frac{0.9 \times 600}{10000} + 0.1 \times 600 = 0.054 + 60 = 60.054 \text{ s}.$$

The speedup becomes 600/60.054 = 9.99. Note that as the CPU becomes faster, the speedup approaches but never exceeds $1/f = 1/0.1 = 10$.

Amdahl's law offers two advices for system design.

- *Optimize the common case*. In the above example, the common case of the old system is CPU processing, which takes 90% of execution time. The other portion for accessing memory and I/O devices forms the uncommon case, which takes only 10% of execution time. So, system enhancement, or system optimization, should focus on the common case: CPU processing.
- *Chase the bottleneck*. After making the CPU 9 times faster, i.e., reducing the CPU processing time to one tenth of the old CPU processing time, the total time becomes 114 s. Furthermore, accessing memory and I/O devices becomes the common case (also called the **bottleneck**). We should change our optimization target to reducing the time for accessing memory and I/O devices. When the bottleneck changes, so does our target. This is called to *chase the bottleneck*.

5.4.4.1 Instruction Pipeline Revisited

Consider again the Fibonacci Computer example in Sect. 2.3, especially the execution of the MOV 0, R1 instruction in a 3-stage instruction pipeline:

- Instruction Fetch (IF): IR←M[PC]
- Instruction Decode (ID): Signals = Decode(IR)
- Instruction Execute (EX): R1 ← 0; PC ← PC+1

Before the instruction is executed, the computer has the configuration (also known as *state*) shown in Fig. 5.35. Note that several new components of the processor are exposed. These system components are invisible to programmers.

- **IR** is the **Instruction Register**, which holds the instruction being executed.
- **MAR** is the **Memory Address Register**, which holds the memory address to be used to access the memory.
- **MDR** is the **Memory Data Register**, which holds the data value for a memory access (load or store).
- Controller is the control circuitry used to generate control signals of instruction.

After the Instruction Fetch (IF) stage finishes, the computer transitions to the configuration in Fig. 5.36a. After the Instruction Decode (ID) stage finishes, IR guides the controller to generate control signals shown in red in Fig. 5.36a. Figure 5.36b shows the configuration after the EX stage finishes. Note that PC has changed to 6, ready to execute the next instruction, MOV R1, M[R0+R2*8-16].

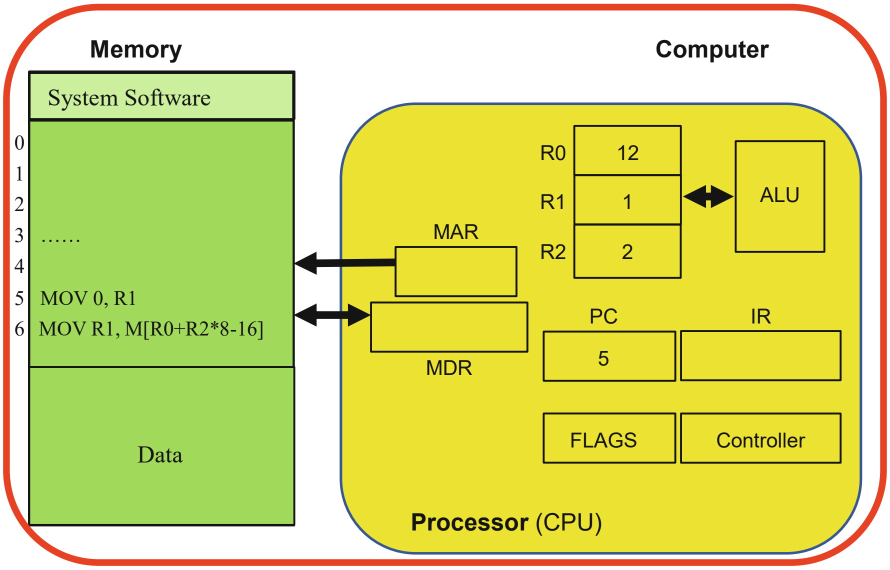


Fig. 5.35 Computer configuration right before the MOV 0, R1 instruction is executed

The instruction pipeline hardware resource can be shared by multiple instruction executions, through the overlapping of pipeline stages, as shown in Fig. 5.37. When there is no overlapping, pipelining has no speed advantage. Suppose a processor has a clock frequency of 1 GHz without overlapping. It can execute 1 billion instructions per second. With overlapping, the processor's clock frequency can increase to 3 GHz. It can now execute 3 billion instructions per second.

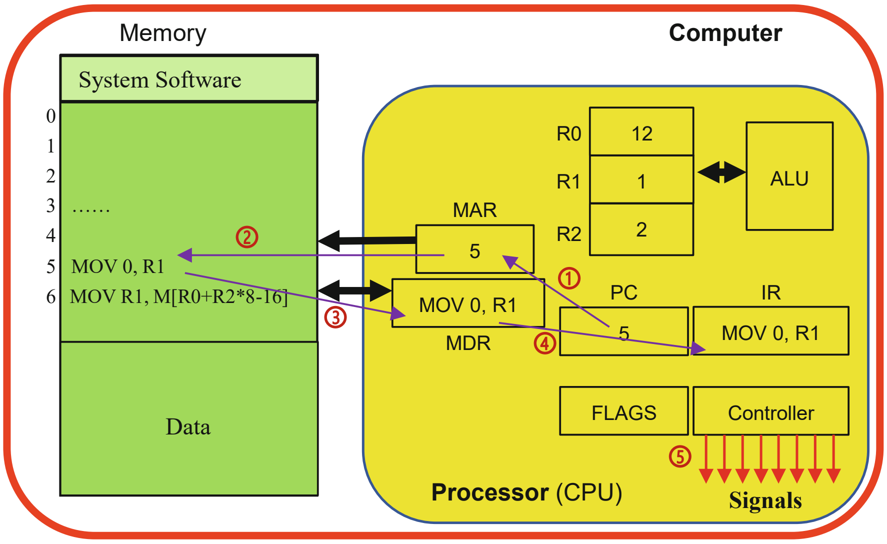
5.4.4.2 Cache

Caching is a method to enhance performance by adding a fast but small memory, called **cache**, between the processor and the memory, as shown in Fig. 5.38.

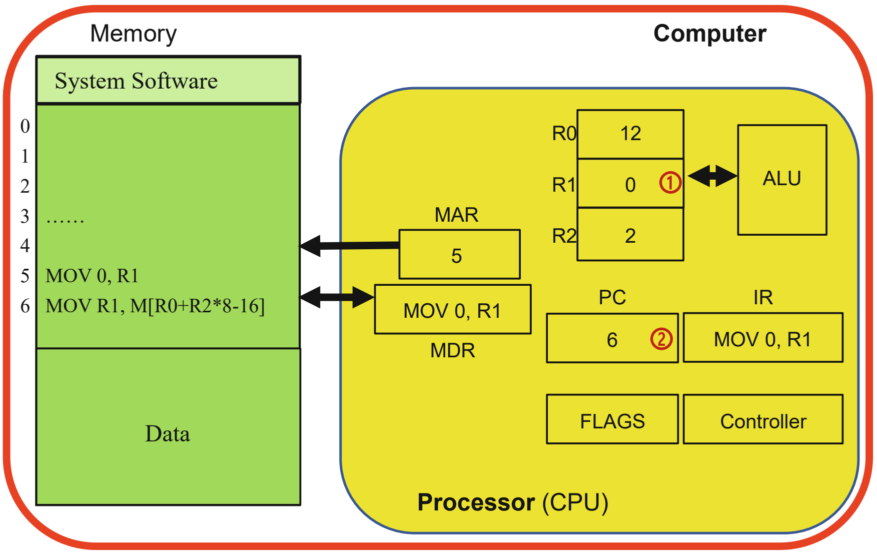
We note a phenomenon that is widely present in computers today. There is a big gap between the processor speed and the memory access speed. Theoretically, a pipelined processor can execute one instruction per clock cycle. However, to perform one memory access (load or store) needs 14 ns, or 43 clock cycles. This disparity is often called the **von Neumann bottleneck**: the memory is too slow to feed data to the processor.

Figure 5.38 uses a hypothetical Fibonacci Computer to show that caching can alleviate this bottleneck and increase performance by utilizing **locality**: data or instructions recently used or nearby are likely to be used again. We store and reuse frequently used data and instructions in the cache.

Cache can itself be layered. In Fig. 5.38b, the cache is organized as two layers. Layer 1 consists of separate **instruction cache** and **data cache**, each holding 32KB



(a)



(b)

Fig. 5.36 Computer configurations after the IF, ID, and EX stages finish, respectively. (**a**) After the Instruction Fetch stage (micro operations ① ② ③ ④), and after the Instruction Decode stage (micro operation ⑤). (**b**) After the Instruction Execute stage (micro operations ① ②)

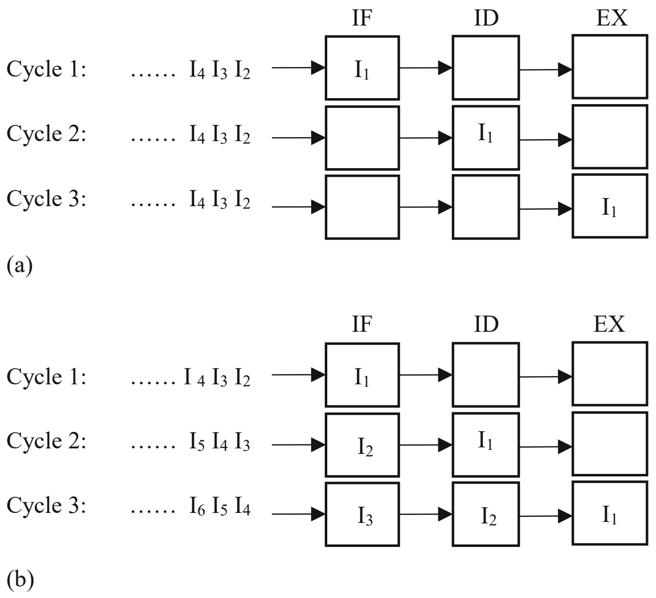


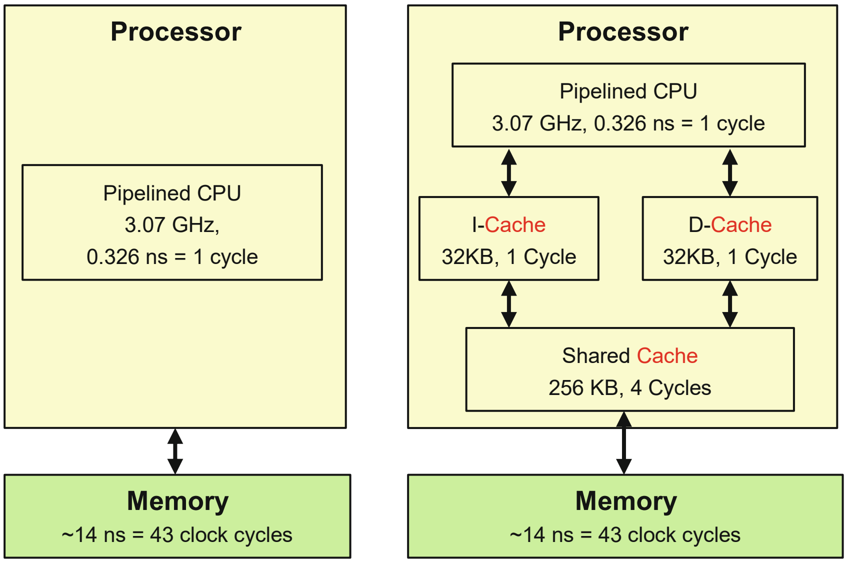
Fig. 5.37 A 3-stage instruction pipeline, without or with overlapping

information but is as fast as the CPU. Layer 2 is a single cache shared by both instructions and data. It is larger (256 KB) but slower (4 cycles) than layer 1.

Example 5.22. Caching Enhances Computer Performance
Let us use the above Fibonacci computer to compute F(92). The loop code showing in red in Fig. 5.38c is executed 92 times, while the code in black only executes once. We thus focus on the seven instructions in the loop code to estimate performance, ignoring all initialization overheads.

The first instruction of the loop, MOV 0, R1, needs to access the memory once, to load the instruction, which needs 14 ns, or 14/0.326 = 43 clock cycles. All other micro-operations are internal operations. We assume that altogether they can be accomplished in one clock cycle. Due to pipeline overlapping, the MOV 0, R1 instruction itself costs 43 clock cycles. The same analysis goes for the fifth, sixth, seventh instructions, each costing 43 clock cycles. Similar analysis applies to the second, third, and fourth instructions. They each need 86 clock cycles due to needing two memory accesses, one for fetching the instruction, and the other for loading/ storing data.

The **peak speed** of a computer is the maximal theoretical number of instructions executed per second. As the joke goes, the vendor of the computer guarantees that you can never exceed this speed. The above Fibonacci computer has a clock frequency of 3.07 GHz, translating to a peak speed of 3.07 GIPS, or Giga instructions per second. The **sustained speed** of the computer is the actual number of instructions executed per second. For the loop code, each iteration of the seven



(a) (b)

		No Caching	Caching
	MOV 0, R1	43	
	MOV R1, M[R0]	86	
	MOV 1, R1	43	
	MOV R1, M[R0+8]	86	
	MOV 2, R2	43	
Loop:	MOV 0, R1	43	1
	ADD M[R0+R2*8-16], R1	86	1
	ADD M[R0+R2*8-8], R1	86	2
	MOV R1, M[R0+R2*8-0]	86	2
	INC R2	43	1
	CMP 51, R2	43	2
	JL Loop	43	3

(c)

Fig. 5.38 Using cache to enhance performance. (**a**) A pipelined computer without cache. (**b**) A pipelined computer with cache. (**c**) Number of clock cycles needed to execute each instruction: without cache vs. with cache

instructions needs 43*4 + 86*3 = 43*10 = 430 clock cycles = 430 * 0.326 = 140 ns. The sustained speed is 7/140 = 0.05 GIPS.

Efficiency is (sustained speed) / (peak speed), which in this case is 0.05 / 3.07 = 1.63%. In other words, the von Neumann bottleneck renders the computer inefficient, only achieving 1.63% of the processor's peak speed.

Now caching comes to the rescue. All instructions and data can be stored in the I-cache and the D-cache, respectively. Thus, accessing memory only need one clock cycle. If the instruction pipeline works smoothly in full overlapping, each instruction will cost only 1 clock cycle.

However, there are **dependencies**, which cause the pipeline to stall. The third instruction ADD M[R0+R2*8-8], R1 actually does R1 + M[R0+R2*8-8] $\rightarrow$ R1, which depends on the result data R1 from the previous instruction. This is called **data dependency**. The pipeline needs to stall for one extra cycle for data to become available. The seventh instruction JL Loop indicates a jump to the back of the Loop, to execute the first instruction MOV 0, R1 again for the next iteration. However, the first instruction MOV 0, R1 cannot be fetched, until the JL Loop finishes. This is called **control dependency**. The pipeline needs to stall for two extra cycles for the jump instruction to finish.

With caching, each iteration of the seven instructions needs 1*3 + 2*3 + 3*1 = 12 cycles = 12 * 0.326 = 3.91 ns. The sustained speed is 7/3.91 = 1.79 GIPS, achieving a speedup of 1.79/0.05 = 35.8. The efficiency becomes 1.79 / 3.07 = 58.32%.

To summarize, caching does not necessarily increase peak speed, but can significantly increase sustained speed and efficiency.

5.4.4.3 Parallel Computing

Another approach to enhancing performance is **parallel computing**, which employs multiple processors to execute a computational task. Each processor in such a **parallel computer** is called a **core**. Most examples of this book only use single-core computers, also known as **sequential computers**. However, students need to know that most real computers today, from smartphones to supercomputers, employ multi-core processors.

Example 5.23. Parallel Computing Enhances Supercomputer Performance
Top500 is a list of the world's fastest supercomputers, maintained since 1993. The performance of a supercomputer is tested by executing a benchmark program, called Linpack, for solving a system of linear equations using Gaussian elimination. In other words, it finds $\boldsymbol{x}$ in $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is an $N \times N$ matrix, and $\boldsymbol{x}$, $\boldsymbol{b}$ are two $N -$ dimensional vectors. The equation $\boldsymbol{Ax} = \boldsymbol{b}$ can be explicitly written as the following, when $N = 3$:

Table 5.9 Parallel computing enhances performance: Top500 Linpack test comparison

Time of test	1993	2020	1993–2020 Growth factor
Top-1 Name	Thinking Machine CM-5	Fujitsu Fugaku	N/A
Problem Size	N = 52,224	N = 20,459,520	392
Speed	59.7 GFlop/s	415,530 TFlop/s	6,960,302
Clock Frequency	32 MHz	2.2 GHz	69
Parallelism	1024 cores	7,299,072 cores	7128
Memory	32 GB	4,866,048 GB	152,064
Power	96.5 KW	28,334.5 KW	294
Cost	US $30 million	US $1 billion	33

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Table 5.9. compares two parallel computers, i.e., the Top-1 systems in 1993 and in 2020, respectively. Note that the speed (performance) increased nearly 7 million times in 27 years, from 59.7 GFlop/s in 1993 to 415 PFlop/s in 2020. This is equivalent to an annual growth rate of more than 24%.

Faster computers can solve bigger problems. The problem size N, i.e., the size of matrix $\boldsymbol{A}$, increased 391 times. Since the number of operations is $O(N^3)$, the number of 64-bit floating-point operations increased more than 60 million times.

The most important factor contributing to this multi-million-fold enhancement is parallelism, which increased over seven thousand times. In contrast, the clock frequency only increased 68 times.

5.5 Exercises

1. Regarding the objectives of systems thinking, which of the following statements are/is correct?
   (a) Systems thinking aims to coping with complexity, meaning that it reduces complexity from $O(N^4)$ to $O(\log N)$, where N is the problem size.
   (b) Systems thinking aims to being thorough, meaning that it considers all necessary and unnecessary details of the system, leaving no stones unturned.
   (c) Systems thinking emphasize flexibility, by designing a specific system for a target application scenario in an ad hoc manner.
   (d) Systems thinking uses abstractions to cope with complexity.
2. Consider the big endian vs. little endian representations of numbers. Which of the following statements are/is correct?

(a) Big endian places the least significant byte in the smallest address.
(b) Big endian places the most significant byte in the smallest address.
(c) Big endian is better than little endian.
(d) Neither the big endian nor the little endian representation is better than the other representation.

3. Which of the following list orders abstractions from low-level to high-level?

(a) Computer, transistor, motherboard, CPU microchip
(b) Computer, CPU microchip, transistor, motherboard
(c) Transistor, CPU microchip, motherboard, computer
(d) Transistor, motherboard, CPU microchip, computer

4. Which of the following list orders abstractions from high-level to low-level?

(a) Computer, transistor, logic gate, memory
(b) Computer, logic gate, memory, transistor
(c) Computer, memory, logic gate, transistor
(d) Transistor, memory, logic gate, computer

5. Which of the following list orders abstractions from high-level to low-level?

(a) Computer, transistor, combinational circuit, sequential circuit
(b) Computer, sequential circuit, combinational circuit, transistor
(c) Combinational circuit, computer, transistor, sequential circuit
(d) Transistor, sequential circuit, combinational circuit, computer

6. The operating system of a computer uses one abstraction to manage all application software programs. What is it?

(a) Instruction
(b) Program
(c) Code
(d) Process

7. The operating system of a computer uses one abstraction to manage all application software programs. What is it?

(a) High-level language program
(b) Machine code program
(c) Processor
(d) Process

8. In a positional number system, the two 6's in 0x06F6 denote different values, as they are in different positions. Which of the following statements are/is correct?

(a) The leftmost 6 denotes 6_{10} and the rightmost 6 denotes 1536_{10}.
(b) The leftmost 6 denotes 1536_{10} and the rightmost 6 denotes 6_{10}.
(c) The leftmost 6 denotes 6×16^2 and the rightmost 6 denotes 6×16^0.
(d) The leftmost 6 denotes six hundreds and the rightmost 6 denotes six ones.

9. Consider the hexadecimal number a = 0x06F6 in a positional number system. Which of the following statements are/is correct?

   (a) The equivalent binary representation is 0000011011110110.
   (b) The value of 0x06F6 is $\sum_{i=0}^{3}\left(a_i \times 16^i\right) = 6 \times 16^2 + 15 \times 16 + 6 = 1782$.
   (c) The base is 16 and the digit set is {0, 1, 2, …, E, F}.
   (d) The base is 10 and the digit set is {0, 1}, since computer only uses binary numbers of 0 and 1.

10. In IEEE 754 floating-point single precision format, the string of 32 bits 01000000010010010000111111011011 represents the decimal value 3.1415927. Assume a string S = 11000000010000000000000000000000 of 32 bits are given. Which of the following statements are/is correct?

    (a) String S is a negative number.
    (b) String S is a positive number.
    (c) String S has an exponent value of 10000000 = 128_{10}.
    (d) String S has an exponent value of 1.
    (e) String S has a significant value of 0.1 = 0.5_{10}.
    (f) String S has a significant value of 1.1 = 1.5_{10}.

11. In IEEE 754 floating-point single precision format, the string of 32 bits 01000000010010010000111111011011 represents the decimal value 3.1415927. Assume a string S = 11000000010000000000000000000000 of 32 bits are given. Which of the following statements are/is correct?

    (a) String S represents the decimal value $0.5{\times}2^{128}$.
    (b) String S represents the decimal value $-0.5{\times}2^{128}$.
    (c) String S represents the decimal value $0.5{\times}2^{1}$.
    (d) String S represents the decimal value $-0.5{\times}2^{1}$.
    (e) String S represents the decimal value $1.5{\times}2^{128}$.
    (f) String S represents the decimal value $-1.5{\times}2^{128}$.
    (g) String S represents the decimal value $1.5{\times}2^{1}$.
    (h) String S represents the decimal value $-1.5{\times}2^{1}$.

12. Consider representing 0 as an IEEE 754 floating-point number in single precision format. Which of the following statements are/is correct?

    (a) The representation is 00000000000000000000000000000000.
    (b) The representation is 10000000000000000000000000000000.
    (c) The representation is 00000000010000000000000000000000.
    (d) The representation is 10000000010000000000000000000000.

13. Floating-point numbers are often approximate values of real numbers. Suppose we want to test whether two floating-point variables X and Y have equal values. We may choose the following Go code:

```
(A) if X==Y { ... }
(B) if math.Abs(X-Y) < math.Pow(10,-15) { ... }
```

Which of the above code are/is correct?

(a) Only A.
(b) Only B.
(c) Either A or B.
(d) Neither A nor B.

14. Mathematically, 2×2=4 and 0.1×0.1=0.01. This is not always true in cyberspace. What does the following code output?

```
X := 2
Y := 0.1
fmt.Println(X*X == 4, Y*Y == 0.01)
```

(a) false false
(b) false true
(c) true false
(d) true true

15. Assume the following code is given.

```
var S [5]byte = [5]byte{'H','E','L','L','O'}
var byteSlice []byte = S[1:2]
fmt.Printf("%s %d", byteSlice, len(byteSlice))
```

The output is:

(a) HE 2
(b) EL 2
(c) H 1
(d) E 1
(e) L 1

16. Assume the following code is given.

```
X := 53
P := &X
fmt.Println(*P)
```

Which of the following statements are/is correct?

(a) X is an integer variable.
(b) Expression &X returns the address value of variable X.
(c) P is a pointer variable holding the address of variable X.
(d) Expression *P returns the value of variable X, i.e., 53.

17. Assume the following code is given.

```
X := 53
p := &X
fmt.Println(*p)
```

The output is:

(a) 5
(b) 3
(c) 53
(d) A hexadecimal number representing variable X's address

18. Write a Go program to invert the first bit of 00111111 to obtain 10111111.
19. Write a Go program to replace the most significant two bits of 00111111 with 01, to obtain 01111111.
20. Refer to the image file Autumn.bmp. Which of the following is NOT metadata?

(a) The photographer's name Li Chundian
(b) The access permission -rw-rw-rw-, i.e., 0666
(c) The size of the file
(d) The pixels of the image

21. Refer to the image file Autumn.bmp. Which of the following is NOT metadata?

(a) The time the file was last modified
(b) The RGB information of the picture elements
(c) The BMP File Header
(d) The BMP Info Header

22. Refer to Example 5.12. The textbook says: the first character is t[0]='H', which is hidden in p[86:90]. Now let us consider the fourth character, which of the following statement is correct?

(a) The fourth character is t[3]='L', which is hidden in p[89:93].
(b) The fourth character is t[3]='L', which is hidden in p[90:94].
(c) The fourth character is t[3]='L', which is hidden in p[98:102].
(d) The fourth character is t[4]='E', which is hidden in p[98:102].

23. Modify program hide-0.go to realize the effect of Fig. 5.13. That is, hide the contents of the text hamlet.txt in the most significant two bits of the pixel array of the image file Autumn.bmp.
24. Precedence, sequencing, selection, and loop are four control abstractions. Which of the following refers to loop?

(a) Check a Boolean condition to determine which part of code to execute next.
(b) Execute one statement after another in the syntactical order of the code.

(c) Evaluate an expression in the order given by the precedence of operators or parentheses.
(d) Repetitive execution of a body of code for a specific number of times.

25. Assume the following code is given, where array X=[1–3]

```
sum := 0
for i := 0; i < 3; i++ {
  sum = sum + X[i]
}
fmt.Println(sum)
```

If the code "for i = 0; i < 3; i++" is replaced by the following code C, what is the output? Put the correct capital letter in the parentheses of each line below.

(a) When C is "for i = 0; i < 1; i++", the output is ()	V: compiling error
(b) When C is "for ; i < 3; i++", the output is ()	W: 1
(c) When C is "for i = 0; ; i++", the output is ()	X: runtime error
(d) When C is "for i = 0; i < 3; ", the output is ()	Y: no termination
(e) When C is "for ; ; ", the output is ()	Z: 6

26. Refer to the recursive program fib-5.go to compute F(5). How many times is the function fibonacci called in executing fib-5.go?
(a) 11
(b) 12
(c) 13
(d) 14
(e) 15

27. Refer to modularization in Sect. 5.3, which of the following statements are/is correct?
(a) The interior of a module is invisible to other parts of a system. They can only access a module through the module's interface.
(b) The interior of a module is visible to other parts of a system, so that all modules can cooperate to maximize system performance.
(c) Two modules are normally isolated. They do not have shared components.
(d) Two modules should have shared components.
(e) All of the above four can be used in designing a system. It is up to the system designer to decide.

28. Regarding combinational circuits discussed in Sect. 5.3.1, which of the following statements are/is correct?
(a) A combinational circuit is a number of interconnected logic gates.
(b) A combinational circuit is a number of logic gates interconnected by wires, with no feedback wires.
(c) A combinational circuit is driven by a clock signal.

(d) A flip flop is a combinational circuit, since it consists of a number of interconnected logic gates.

29. How many 1-input-1-output combinational circuits are there? If two combinational circuits realize the same truth table, they are equivalent and counted as one circuit.

    (a) 1
    (b) 2
    (c) 4
    (d) 8

30. How many N-input-1-output combinational circuits are there? If two combinational circuits realize the same truth table, they are equivalent and counted as one circuit.

    (a) N
    (b) N^2
    (c) 2^N
    (d) 2^{2N}

31. Refer to the full adder in Fig. 5.22. Which of the following statements are/is correct?

    (a) A full adder receives three inputs, two operand bits and one carry-in bit, to generate a sum output bit and a carry-out bit.
    (b) A full adder is a 2-input-2-output combinational circuit.
    (c) A full adder is a 3-input-2-output combinational circuit.
    (d) The sum output bit is 1, if an odd number of the three input bits are 1.

32. Refer to the fast adder in Fig. 5.24. Denote $G_0 = X_0 \cdot Y_0$, $G_1 = X_1 \cdot Y_1$, and $P_0 = X_0 \bigoplus Y_0$, $P_1 = X_1 \bigoplus Y_1$. Which of the following equations are correct?

    (a) $C_2 = C_0 \cdot P_0 \cdot P_1 + P_1 \cdot G_0 + G_1$
    (b) $C_2 = C_0 \cdot G_0 \cdot G_1 + G_1 \cdot P_0 + P_1$
    (c) $C_2 = X_1 \cdot Y_1 + (X_1 \bigoplus Y_1) \cdot C_1$
    (d) $C_2 = C_0 \cdot X_0 \cdot Y_0 + X_1 \cdot Y_1 + C_1$

33. Refer to the fast adder in Fig. 5.24. Denote $G_0 = X_0 \cdot Y_0$, $G_1 = X_1 \cdot Y_1$, and $P_0 = X_0 \bigoplus Y_0$, $P_1 = X_1 \bigoplus Y_1$. Which of the following equations are correct?

    (a) $C_2 = X_1 \cdot Y_1 + (X_1 \bigoplus Y_1) \cdot C_1$
    (b) $C_2 = G_1 + P_1 \cdot C_1$
    (c) $C_1 = G_0 + P_0 \cdot C_0$
    (d) $C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$

34. Regarding sequential circuits discussed in Sect. 5.3.2, which of the following statements are/is correct?

(a) Sequential circuits are comprised of combinational circuits and state circuits.
(b) A state circuit is realized by one or more memory cells or flip flops.
(c) A flip flop is a combinational circuit.
(d) A flip flop is an augmented combinational circuit with feedback wires.

35. Regarding sequential circuits discussed in Sect. 5.3.2, which of the following statements are/is correct?

(a) Sequential circuits are comprised of combinational circuits and state circuits.
(b) A state circuit can be realized by one or more delay flip flops.
(c) A sequential circuit in this book is actually a clock-synchronous sequential circuit, since the state circuit is driven by a clock signal.
(d) A sequential circuit in this book is actually a clock-synchronous sequential circuit, since the combinational circuits are driven by a clock signal.

36. Regarding memory circuits discussed in Sect. 5.3.2, which of the following statements are/is correct?

(a) DRAM needs constant refreshing, to compensate for electric leakage.
(b) A DRAM cell needs six transistors.
(c) A 1KB DRAM memory contains over 8000 capacitors.
(d) A 1KB SRAM memory contains over 8000 capacitors.
(e) ROM is a type of non-volatile memory.
(f) SRAM does not need refreshing, meaning its content is not lost when the power is turned off.

37. Consider the characteristics of memory circuits discussed in Sect. 5.3.2. Put the correct capital letter in the parentheses of each line below.

(a) Volatile, fast, expensive correspond to ()	U: DRAM
(b) Volatile, slow, inexpensive correspond to ()	V: Read-Write NVM
(c) Nonvolatile, read-only correspond to ()	W: ROM
(d) Nonvolatile, read-write correspond to ()	X: SRAM

38. Refer to Example 5.16. Design a 4-bit serial subtractor and verify its correctness by executing $11 - 9 = 2$ and $9 - 11 = -2$.
39. Refer to Example 5.16. Design an n-bit serial subtractor and verify its correctness by executing 99. . .9 – 99. . .9 and see that the result is indeed 0.
40. Refer to Example 5.17. Design the instruction set architecture for an Accumulator Computer to execute the following computation, assuming X=[1, 2, 3,. . ., *N*]

```
sum := 0
for i := 0; i < N; i++ {
  sum = sum + X[i]
}
```

41. Regarding the software stack used in this book, which of the following statements are/is correct?
    (a) The Go compiler is a piece of firmware.
    (b) The Go compiler is a piece of system software.
    (c) The Linux Shell is a piece of application software.
    (d) The Linux Shell is a piece of middleware.
42. Regarding the software stack used in this book, which of the following statements are/is correct?
    (a) The Web browser is a piece of firmware.
    (b) The Web browser is a piece of middleware.
    (c) The Linux operating system is a piece of system software.
    (d) The Linux operating system is a piece of application software.
43. Regarding the cycle principle in Sect. 5.4, which of the following statements are/is correct?
    (a) An instruction cycle consists of multiple program cycles.
    (b) An instruction cycle consists of multiple clock cycles.
    (c) A program cycle consists of multiple instruction cycles.
    (d) A program cycle consists of multiple clock cycles.
44. Regarding the robustness principle in Sect. 5.4, which of the following statements are/is correct?
    (a) Be conservative in what you do to others, be liberal in what you accept from others.
    (b) Be conservative in what you accept from others, be liberal in what you do to others.
    (c) Be tolerant of inputs, be strict on outputs.
    (d) Be tolerant of outputs, be strict on inputs.
45. After a jump instruction finishes, the destination instruction should be executed. What happens when the jump instruction finishes?
    (a) The destination instruction is assigned to the program counter PC.
    (b) The address of the destination instruction is assigned to the program counter PC.
    (c) The destination instruction is assigned to the instruction register IR.
    (d) The address of the destination instruction is assigned to the instruction register IR.
46. After a jump instruction finishes, the destination instruction should be executed. What happens when jump instruction finishes?
    (a) The address of the jump instruction is assigned to the program counter PC.
    (b) The address of the destination instruction is assigned to the program counter PC.

(c) The address of the destination instruction is assigned to the instruction register IR.
(d) The difference of the addresses of the destination instruction and of the jump instruction is assigned to the instruction register IR.

47. When the following event happens, what exception occurs? Put the correct capital letter in the parentheses of each line below. App exception denotes an exception that occurs in the application program that does not cause hardware error, interrupt or machine check.

(a) Execute a "divide by zero" operation ()	W: App exception
(b) ID stage sees an undefined opcode ()	X: Hardware Error
(c) CPU knows something is wrong but not what ()	Y: Interrupt
(d) The user moves the mouse ()	Z: Machine Check

48. When the Fibonacci Computer is executing code and the following situation happens, what exception occurs? Put the correct capital letter in the parentheses of each line below.

(a) The code does not terminate ()	W: Hardware Error
(b) ID stage sees opcode 111 ()	X: Interrupt
(c) The power is turned off ()	Y: Machine Check
(d) The user clicks the mouse ()	Z: No exception

49. Suppose the execution time of a program is 100 s, of which 80% is spent on CPU processing, and 20% on accessing memory and the hard disk. Assume the CPU clock frequency increases from 1 GHz to 1 THz, which is an enhancement of about 1000 times. What is the execution time of the program on this enhanced computer?

(a) 0.02 s
(b) 0.2 s
(c) 2 s
(d) 20 s
(e) 200 s

5.6 Bibliographic Notes

The chapter quotation by Confucius (孔子) is from the classic 论语, or *Analects*. The original Chinese text is "民可使由之不可使知之", which has multiple interpretations. This particular translation from the abstraction perspective benefited from discussions with Professor Jinshu Zhu of Xihua University. The quotation by Leslie Lamport is from [1]. Luszczek, *et al*, presents the idea of using a few smartly design benchmarks to represent many applications [2]. Bergman presents the Bergman number system [3]. Discussion on Fibonacci number systems can be found in [4]. The IEEE floating-point number standard is discussed in [5, 6]. Alkhatib, *et al*, presents a vision of seamless intelligence of the IEEE Computer Society

[7]. The English translation of Head Zhou, quoted in Sect. 5.4.1, is from [8]. Postel proposed the principle of robustness in [9]. John von Neumann proposed the exhaustiveness principle in 1945. A reprinting can be found in [10]. Amdahl presented the idea of Amdahl's law in [11], although no formula can be found in that short paper. TOP500 is a website maintaining performance data on the world's 500 fastest computers [12].

References

1. Lamport L (2018) If you're not writing a program, don't use a programming language. https://www.heidelberg-laureate-forum.org/video/lecture-if-youre-not-writing-a-program-dont-use-a-programming-language.html
2. Luszczek PR, Bailey DH, Dongarra JJ, Kepner J, Lucas RF, Rabenseifner R, Takahashi D (2006) The HPC Challenge (HPCC) benchmark suite. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, vol. 213, no. 10.1145, pp. 1188455–1188677
3. Bergman G (1957) A number system with an irrational base. Math Mag 31(2):98–110
4. Ahlbach C, Usatine J, Frougny C, Pippenger N (2013) Efficient algorithms for Zeckendorf Arithmetic. Fibonacci Q 51(3):249–255
5. Severance C (1998) IEEE 754: an interview with William Kahan. Computer 31(3):114–115
6. Kahan W (1996) IEEE standard 754 for binary floating-point arithmetic. Lecture Notes on the Status of IEEE, vol 754, no 94720-1776, p 11
7. Alkhatib H, Faraboschi P, Frachtenberg E, Kasahara H, Lange D, Laplante P et al (2015) What will 2022 look like? The IEEE CS 2022 report. Computer 48(3):68–76
8. Nylan M (1993) The Canon of supreme mystery by Yang Hsiung: a translation with commentary of the T'ai Hsuan Ching. SUNY Press, Albany
9. Postel J (1980) DoD Standard–Transmission Control Protocol–RFC 761. University of Southern California. https://datatracker.ietf.org/doc/rfc761/
10. Von Neumann J (1993) First draft of a report on the EDVAC. IEEE Ann Hist Comput 15(4):27–75
11. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, spring joint computer conference, pp. 483–485.
12. TOP500. https://www.top500.org Web site

Chapter 6
Network Thinking

[The users of the World Wide Web] felt they were part of a system, a new system of humanity collectively producing more and more value.
Everybody would find mutual understanding with each other across the Web, rather than fighting each other, sending each other's sons out to fight each other in the fields.
—Tim Berners-Lee, 2019

This chapter extends the principles in the previous chapters to networks, or more precisely, computer networks. We use the familiar global Internet as the main example. The **Internet** is a global *internetwork* (network of networks) connecting many local networks and computers through the Internet Protocol (IP).

Network thinking studies and utilizes the networking aspect of computing to generate computational and societal values. It has two basic concepts: *connectivity* and *protocol stack*.

Connectivity refers to what and how nodes of a network are connected. We pay special attention to two knowledge units: *naming* and *topology*. What nodes are in the network? How to name them? How to find a specific node? How are the nodes interconnected? Does the network structure change over time? Is the network topology static, dynamic, or evolutionary? Students learn how to use a user-friendly Web URL name, to access a web resource on a specific computer.

A **protocol stack** is a stack of layers of rules governing communication. It is used to enable multiple nodes of a network to communicate with one another, in order to automatically collaborate in accomplishing a common computational task. We introduce how the Web over Internet protocol stack works. Key concepts involved include: message passing, packet switching, interfaces of protocol layers, switch and router. We also prepare students to do the Personal Artifact project by introducing Web programming examples.

In addition, this chapter introduces network laws and responsible computing, as the societal impact is most obvious on the Internet. We discuss *network effect* as exemplified by Metcalfe's law and the viral marketing phenomenon. Examples are used to illustrate issues on security, privacy, and professional code of conduct.

Z. Xu, J. Zhang, *Computational Thinking: A Perspective on Computer Science*,
https://doi.org/10.1007/978-981-16-3848-0_6

6.1 Network Terms

Often, a computational process involves not just one entity, but also a group of interconnected entities, which may refer to or communicate with one another. Here, an entity may be an abstract or real entity regarding a computer, a person, or a thing. Here, a thing could be a physical thing, a document, an idea, or a concept.

A group of interconnected entities is called a **network**. The entities are called the **nodes** of the network. For instance, all the one billion WeChat users form a network with one billion nodes. As another example, all articles of computing form a network of computer science literature. The nodes (articles) are interconnected by citations. The nodes may refer to one another, but do not necessarily communicate with one another by sending or receiving messages.

A computational process may treat a network as an *object*. For instance, we may design a computational process to compute the topology of the network of WeChat users. Then, the network is an input object, and the topology of the network is an output object. A network may also be the *subject* of a computational process. For example, a network of computers may be used to execute the above task of topology computation: the network computes the topology.

Let us start with some terms of networking, by looking at parts of networks of three universities, shown in Fig. 6.1.

A **local area network** (LAN) connects devices (shown as squares in Fig. 6.1) in the same building or even on the same campus. A **metropolitan area network** (MAN) connects LANs in the same city. A **wide area network** (WAN) spans multiple cities or even multiple countries. These are demonstrated in Fig. 6.1.

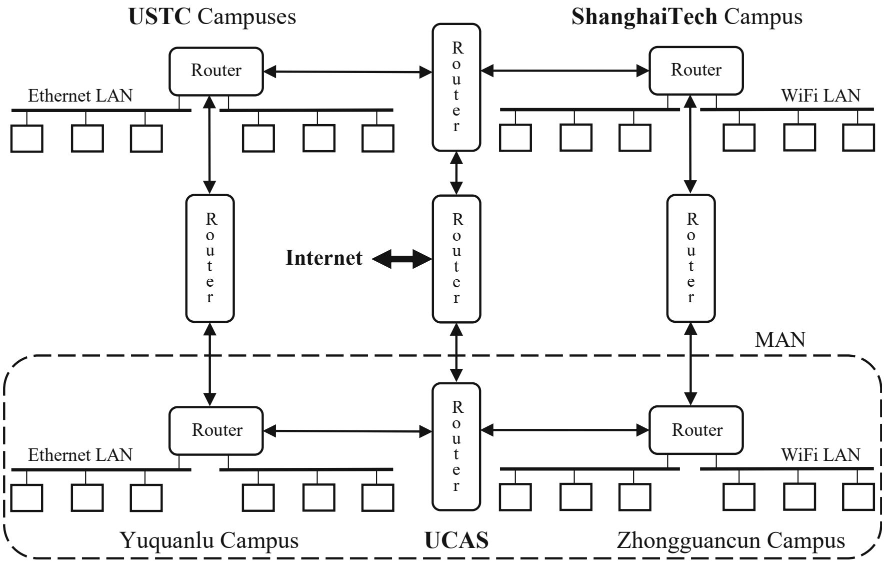


Fig. 6.1 Illustrating parts of networks of UCAS and sister universities

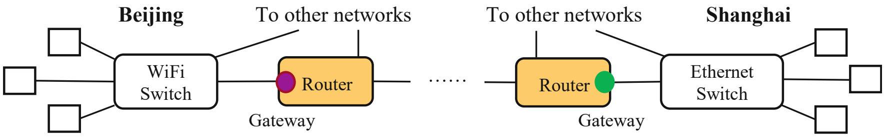


Fig. 6.2 Connecting two heterogeneous campus networks by routers

The University of Chinese Academy of Sciences (UCAS), the University of Science and Technology of China (USTC) and the ShanghaiTech University (ShanghaiTech) have campuses in Beijing, Hefei and Shanghai, respectively. Their LANs and MANs are connected by a WAN called the China Science and Technology Network (CSTNet), which is connected to and forms a part of the global Internet.

An institution providing Internet connection services is called an Internet service provider (**ISP**). The institution running CSTNet, also called CSTNET, is the ISP for all students, faculties, and staffs of Chinese Academy of Sciences.

Suppose a student, Zhang Lei, is studying in a classroom on the Zhongguancun campus of UCAS in Beijing. She wants to use her laptop computer to access a server located in a machine room on the ShanghaiTech campus. Let us trace the connections and see the network terms involved.

- **Host**. Any device connected to a network is called a host (shown as a square in Fig. 6.1). It could be a laptop computer, a printer, a server, or an environment sensor. A host directly used by the user is called a **client**, which makes requests to be served by another host called **server**. Zhang's laptop computer is a client host, and the server on the ShanghaiTech campus is a server host. Note that hosts do NOT include **networking devices** discussed below, such as access point, hub, switch, router, and gateway devices.
- **Access Point** (AP). Zhang uses a wireless protocol called WiFi to access the Internet. Her laptop computer first needs to connect to an **access point** (AP). The AP device converts wireless signals to wired signals and vice versa. The AP device is part of the WiFi LAN and not explicitly shown in Fig. 6.1.
- **Hub** and **Switch**. A small LAN is often a bus structure realized by connecting a few hosts to a central device called a network *hub*. Multiple small LANs can be connected to form a bigger LAN to accommodate more hosts or to extend the distance covered. A network *switch* connects multiple hosts or LANs. The same protocol, e.g., Ethernet, must be used within a LAN, small or big. In such a *homogeneous* network, hosts are said to be *homogeneously connected*.
- **Router**. What if we want to connect several networks that use different protocols? Networking devices called **routers** are used to solve this *internetworking* problem, i.e., connecting multiple member networks into a bigger *heterogeneous* network, e.g., the Internet. Each member network of the Internet may internally use a different networking technology of its own choice, such as WiFi, Ethernet, or Infiniband. Figure 6.2 shows how multiple routers (the brown boxes) connect

two heterogeneous member networks: the UCAS Beijing campus network using WiFi, and the ShanghaiTech campus network using Ethernet.
- A **gateway** of a member network is the router connecting the member network to a bigger network, e.g., Internet. In Fig. 6.2, the purple and green dots indicate gateways for the Beijing network and the Shanghai network, respectively.

6.2 Connectivity

Connectivity refers to the structure of a network. Only when an entity is connected, i.e., having become a node of a network, can it be accessed. Connectivity studies problems related to *naming* and *topology*, such as the following:

- How to name the nodes of a network? How to find a specific node? How to refer to a specific node?
- How are the nodes interconnected? Does the network structure change over time? Is the network static, dynamic, or evolutionary?

It is customary to use a mathematical structure, called a **graph**, to represent a network. A graph $G = \langle V, E \rangle$ is a pair of sets. Here, $V = \{v_1, v_2, \ldots, v_n\}$ is the set of nodes. A node is also called a **vertex**. $E = \{e_1, e_2, \ldots, e_m\}$ is the set of **edges** or *links*. Each edge connects two nodes. Figure 6.3 shows two networks. A directed edge shows a one-way connection. An undirected edge is a two-way connection. Note that node v_1 connects to itself, while v_3 does not connect to any node.

6.2.1 *Naming*

We specify the nodes of a network by **naming**, i.e., giving a name to every node. As illustrated in Table 6.1, every network has a **namespace**, which consists of all names of the network specified by a naming scheme.

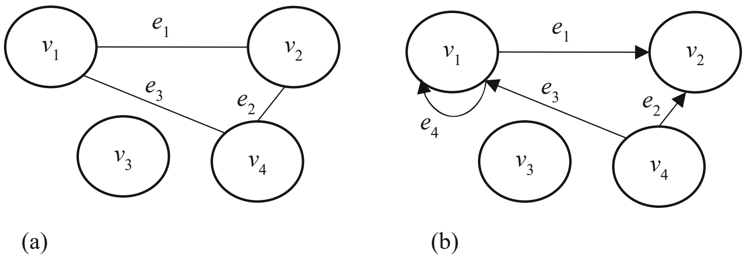


Fig. 6.3 Networks can be represented as directed and undirected graphs. (**a**) An undirected graph, (**b**) A directed graph

Table 6.1 Examples of Naming Schemes

Namespace	Instance	Remark
Personal name	Joan Smith	Personal names in a country
WeChat user	ZhongguanVillager	Any legitimate string per WeChat standard
URL	www.ict.ac.cn/cs101	Universal Resource Locator of a webpage
Internet site	www.ict.ac.cn	Any domain name by IETF standards
Email address	zxu@ict.ac.cn	userName@domainName
IP address	159.226.97.84	Internet protocol address per IETF standards
Phone number	189-6666-8888	11 decimal digits by Telcom provider standards
MAC address	00-1E-C9-43-24-42	12 hexadecimal digits per IEEE standards

A name is usually a string of characters. Whether a string is legitimate is determined by a naming scheme, which is often specified by some technical **standard**. Three influential standards bodies are IEEE, IETF and W3C. They are all professional volunteer organizations. Ethernet names and protocols are designed by the Institute of Electrical and Electronics Engineers (IEEE). Internet standards (called **RFC**s) are designed by the Internet Engineering Task Force (IETF). Web names and protocols are designed by the World Wide Web Consortium (W3C).

Designing a namespace needs to consider the following issues:

- *Uniqueness*. Does a name map to a unique node? The email address namespace enjoys uniqueness, but the namespace of personal names of a country's population does not have uniqueness. There may be multiple persons named Joan Smith, causing *name conflicts*, which in turn may lead to wrong connections.
- *Autonomy*. Can a user create or change a name on his own? Such autonomy has the advantage of convenience, but may lead to chaos. One may change a URL, i.e., the name of a webpage. However, Web links to the old URL become invalid. For many naming schemes, creating or modifying a name need to go through a centralized process, involving an authority of name registry.
- *Friendliness*. Are the names user-friendly, i.e., understandable by humans? The eight name schemes in Table 6.1 have decreasing user friendliness. "Joan Smith" is much more understandable than "00-1E-C9-43-24-42", which is the name of the network interface circuitry in a computer, also called MAC address.
- *Name conversion*. An entity can have two namespaces. The Internet site with domain name www.ict.ac.cn has an Internet Protocol (IP) address 159.226.97.84. The Domain Name System (**DNS**) converts a domain name to its IP address.

6.2.1.1 The Namespaces of the Internet: Domain Names vs. IP Addresses

Recall that the **Internet** is a global network of networks, connecting many local networks and computers through a common set of protocols, particularly the Internet Protocol (IP).

Let's put ourselves in the shoes of early Internet designers and ask this question: How many names are needed to uniquely denote every node of the global Internet?

There are two types of nodes in the Internet:

- *Internet hosts* are computers connected to the Internet, such as our laptop PCs and smartphones. Here, computers included Internet-connected devices such as printers, TVs, and sensors, which can be viewed as embedded computers.
- *Networking devices* on the Internet, such as network routers and switches.

Every node of the Internet should be named. At a minimum, a node must have an address, called the Internet Protocol address, or IP address. See Box 6.1 for a comparison between names and addresses.

Two types of IP addresses are used today.

- **IPv4 addresses**. An Internet Protocol version 4 address (IPv4 address) occupies 32 bits, organized as a 4-field format xxx.xxx.xxx.xxx such as 159.226.97.84, where each field is normally shown as a decimal number from 0 to 255. This IPv4 address format can generate 2^{32}, or over 4 billion, different IP addresses.
- **IPv6 addresses**. An Internet Protocol version 6 address (IPv6 address) occupies 128 bits. This IPv6 address format can generate 2^{128} different IP addresses.

If we assign a unique IP address to a node of the Internet, the IPv4 addressing scheme can accommodate an Internet of over 4 billion nodes, including Internet hosts and networking devices. In reality, multiple IP addresses can map to one node (cf. RFC 791). Thus, the 32-bit IPv4 addressing scheme implies that the Internet can have fewer than 4 billion nodes, and even fewer host computers. Today, we already have more than 4 billion hosts on the Internet.

Longer address formats could have been used to accommodate more nodes. But, longer address formats also consume more bit resources. To send a message from node A to node B, the message needs to include the source address for A and the destination address for B, besides the payload data of the message. For a short message of 4-byte payload data with IPv4 addresses, we need to send at least 96 bits, of which only 1/3 is payload, and 2/3 is IP address metadata information.

The Internet pioneers did a tradeoff in 1981 (RFC 791) by choosing a 32-bit format for the IPv4 addressing scheme. Note that an IP address assignment to a node is not permanent, and may be recycled. Still, in 2019, all IPv4 addresses worldwide were assigned, leaving no free IPv4 addresses to allocate.

Fortunately, the IETF standards body created a much longer IPv6 address format in 1995 (RFC 1883). If each node of the Internet is assigned a unique IP address, the IPv6 address format can accommodate $2^{128} = 2^{32} \times 2^{96}$ nodes, or 2^{96} times the capacity of the IPv4 scheme.

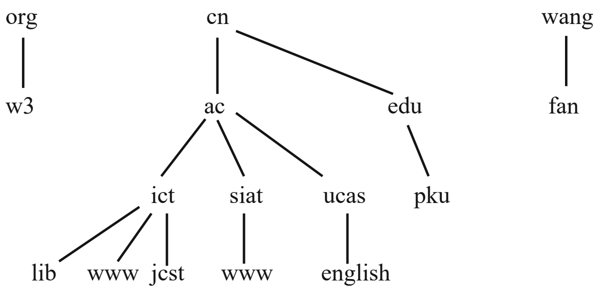


Fig. 6.4 Parts of the global domain name hierarchies

Box 6.1. Addresses vs. Names

A **name** is a string of characters used to denote an entity. **Addresses** are a special form of names. An address can be directly used to access an entity, while a non-address name needs to be first converted into an address before accessing the entity denoted by the name.

For instance, *www.ict.ac.cn* is a name: the **domain name** of the website for the Institute of Computing Technology, Chinese Academy of Sciences. To access this website from a Web browser, what really happens is that this domain name is converted by DNS into the corresponding **IP address** 159.226.97.84. This IP address is then used to access the website. It is difficult to remember an IP address such as 159.226.97.84. Users prefer to use the more user-friendly domain name www.ict.ac.cn.

A computer on the Internet can serve as a *client host* such as a user's PC, a *server host* such as a supercomputer, or a *networking device* such as a router. A computer on the Internet can have one or more IP addresses. Multiple domain names can be mapped to the same IP address. Conversely, the same domain name can be served by multiple computers. A popular website, sometimes called a *hub* node of the Internet, may need dozens of computers to serve requests during peak hours.

Domain names are organized in hierarchies. The structure of the hierarchies can be seen when reading a domain name from right to left. In www.ict.ac.cn, "cn" is the top-level domain by *country*, "ac" is the second-level domain, and "www" is at the lowest level. There are also *generic* top-level domains, such as "com" and "org" in github.com and w3.org. There are over 1000 top-level domain names today. A student named Fan Wang (范望) may register an Internet domain name as fan. wang (Fig. 6.4).

6.2.1.2 The Namespace of the World Wide Web: URL

A *uniform resource locator* (**URL**) is a string that specifies the location of a Web resource for a browser to easily access the resource. A **web resource** can be a website, a webpage, a picture file on the Web, etc. Sometimes, people calls a URL a **web address**. As illustrated in Fig. 6.5, a basic URL consists of three parts, separated by "://" and "/". There can be infinitely many web resources, thus infinitely many URLs.

The first part specifies the *scheme*, which indicates a protocol. The most popular protocol used on the Web is the *hypertext transport protocol* (**HTTP**), and its secure version *hypertext transport protocol secure* (**HTTPS**). This book contains examples of both protocols. Other protocol names include FTP for transferring a file from one computer to another computer, and MAILTO for email.

The second part specifies the *host*, which indicates the name of an Internet host computer. The host can be specified by its domain name or its IP address. When we enter in the browser http://www.ict.ac.cn or http://159.226.97.84, we get the same result, indicating the same host computer.

The third part specifies the *path*, which indicates where the resource is located on the host, by specifying a file path name. Note that the HTTP root directory may be different from the host computer's operating system root directory. In Fig. 6.5, the file path name is /cs101, e.g., indicating that the resource is located at cs101 in the HTTP root directory of the host.

Two special URLs are noteworthy. First, when we enter in the browser a URL of a website without a path (e.g., http://www.ict.ac.cn) or with only the HTTP root (e.g., http://www.ict.ac.cn/), we are accessing the **homepage** of the website.

Second, when we enter in the browser the URL http://localhost, or equivalently http://127.0.0.1/, this special host is reserved for the local computer (or local host), where the browser is executing. The string "localhost" indicates the **loopback domain name** of a computer, and 127.0.0.1 is its **loopback IP address**. Project 4 of this book will use local host extensively to develop and debug webpages.

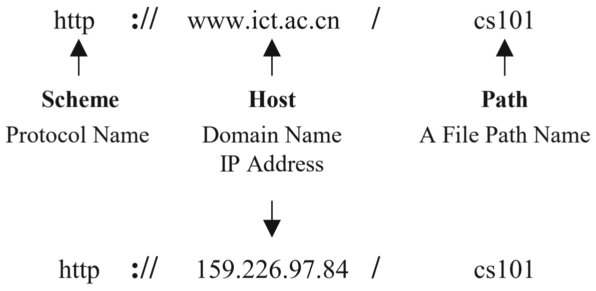


Fig. 6.5 Anatomy of a uniform resource locator (URL)

6.2.2 *Network Topology*

At any moment, the structure of a network is a graph, showing what nodes and edges are in the network, and how the nodes are interconnected by the edges. This graph is also called the **topology** of the network. The topology of a network may change with time. Depending on how the network topology changes, we can classify networks into three classes, as illustrated in Fig. 6.6.

- **Static networks**. Static networks do not change their nodes and edges. More precisely, a network $G = \ < V, E >$ is a static network, if the node set V and the edge set E do not change over time. In Fig. 6.6, the fully connected graph and the star network are both static networks.
- **Dynamic networks**. A dynamic network does not change its nodes, but may change its edges. In Fig. 6.6, the **bus** and the **crossbar** switch networks are both dynamic networks. At one moment, the bus may actually connect the processor (P) and the memory (M). At the next moment, the bus may actually connect the memory (M) and the input device (I). The bus supports a *shared-media network*, while the crossbar supports a *switching network*.
- **Evolutionary networks**. Evolutionary networks change both nodes and edges over time. An example is the network of all webpages on the World Wide Web. The node set V (the set of all webpages) and the edge set E (the set of Web links) both constantly change over time.

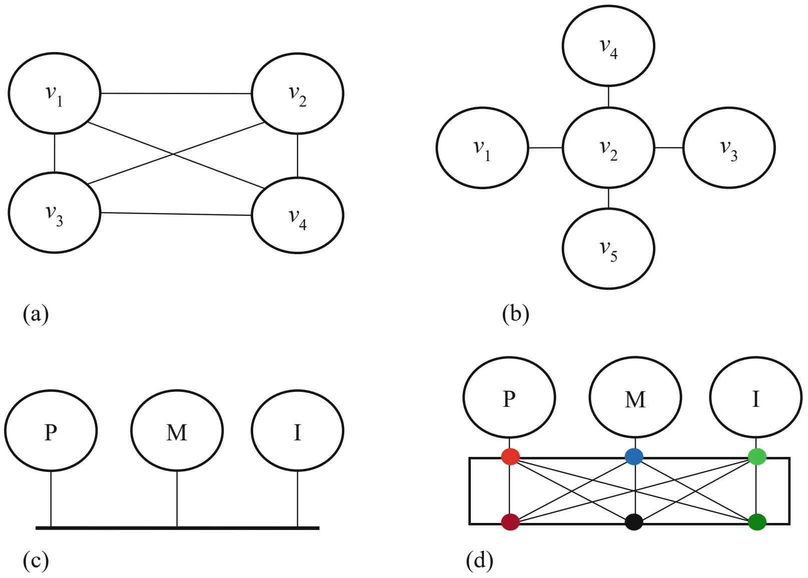


Fig. 6.6 Examples of static and dynamic networks. (**a**) A fully connected graph, (**b**) a star network. Nodes connected by (**c**) a bus or by (**d**) a crossbar switch

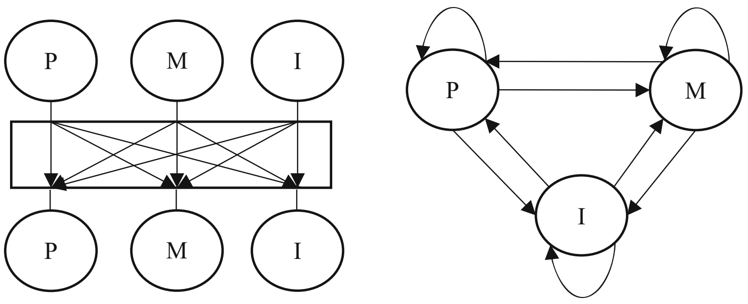


Fig. 6.7 Expanded illustrations of a network of three devices interconnected by a crossbar switch

In Fig. 6.6d, three devices (circles) interconnect to one another through a crossbar switch (rectangle). The three devices are the processor (P), the memory (M) and the input device (I). Each device is connected to both an input port and an output port of the crossbar switch. For instance, the processor (P) is connected to the red input port and the purple output port in Fig. 6.6d. This is redrawn in Fig. 6.7, where both P nodes denote the same processor. A device may be able to send messages and receive messages at the same time. In practice, an input port and an output port can be implemented as one port. Such a port capable of simultaneous bidirectional communication is said to work in the **full-duplex** mode. That is, full-duplexing allows data to flow into and out of a port at the same time.

The crossbar switch is actually a fully connected graph, as shown in Fig. 6.7. At any moment, all or parts of the edges are active. A crossbar switch is more powerful than a bus. It enables all devices to connect to one another at the same time. A bus can only connect two devices at a time, except for **broadcasting**, where a device sends a message to all devices on the bus.

Example 6.1. How Network Thinking Helped Create Modern Search Engines
Network thinking offers a new perspective to look at problems and can lead to innovative solutions. A case in point is search engine design. Early search engines computed search results by matching the key-words in search queries to the contents of webpages (*nodes*). These first-generation search engines only utilized nodes of the network of webpages.

Around 1996, Jon Kleinberg, Robin Li (李彦宏), and Larry Page, independently observed a phenomenon: Web links (*edges*) also significantly influence the relevance of search results. They utilized both nodes and edges to develop the second-generation search engines with much better results. This approach more fully utilizes network thinking and created Google and Baidu companies, serving billions of users and generating annual revenue over $100 billion.

☷

6.3 Protocol Stack

A **protocol** is a set of rules enabling two nodes of a network to communicate with each other, in order to automatically collaborate in accomplishing a common computational task. A **protocol stack** is a stack of layers of protocols which work together. This section introduces the Web over Internet protocol stack, to elaborate the basic concepts of communication in a computer network.

6.3.1 The Web over Internet Stack

A basic innovation of the Internet is to use *packet switching* to replace the *circuit switching* technology used in traditional telecommunication networks.

- **Circuit switching**. When two parties are having a telephone conversation lasting 10 min over a telecommunication network, a physical circuit is established and dedicated to this task during the entire time period of conversation. No other person can use the resource of the circuit for this period of 10 min.
- **Packet switching**. When two parties are having a telephone conversation lasting 10 min over the Internet, a person's utterances are viewed as digital **messages** communicated to the other person. Each message is divided into a number of small **packets**, which are sent together with packets from other people or devices, over the same communication channel (circuit). In other word, the same channel can accommodate multiple telephone conversations in this period of 10 min.

Example 6.2. Comparison of Circuit Switching and Packet Switching
Three students each want to download a file from the Internet over a shared communication channel, as shown in Fig. 6.8. Suppose the communication channel has a bandwidth of 10 Mbps (million bits per second) and all three students start downloading at time 0. How much time does each of these downloading tasks take?

Figure 6.8 shows the Internet as a cloud picture, where three files are stored. The circuit switching approach works as shown in Fig. 6.8a. Suppose student Smith's downloading request is first received by the system. A dedicated, end-to-end communication circuit is established between the file and Smith's PC. Then the system transmits file Autumn.bmp from its source server to the destination computer (Smith's PC). Let us simplify matters by further assume that all overheads are ignored. Then, because each byte has 8 bits, the first downloading task takes

$$T_1 = 9.14 \times 8/10 = 7.31 \text{ s.}$$

In other words, the task to download Autumn.bmp finishes at $T_1 = 7.31$ s.

Similarly, the second task to download hamlet.txt only needs

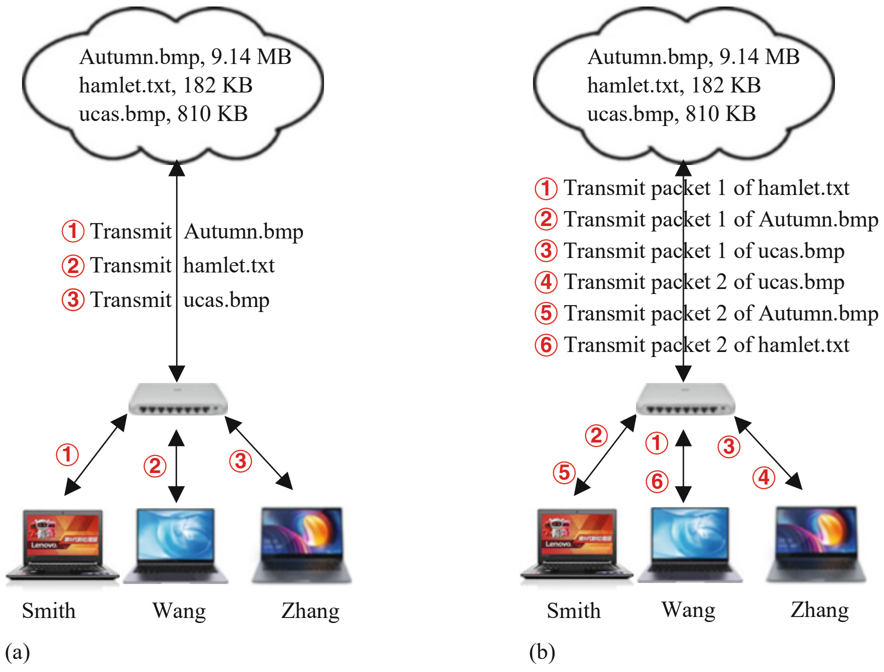

Autumn.bmp, 9.14 MB
hamlet.txt, 182 KB
ucas.bmp, 810 KB
① Transmit Autumn.bmp
② Transmit hamlet.txt
③ Transmit ucas.bmp
Smith
Wang
Zhang
(a)
① Transmit packet 1 of hamlet.txt
② Transmit packet 1 of Autumn.bmp
③ Transmit packet 1 of ucas.bmp
④ Transmit packet 2 of ucas.bmp
⑤ Transmit packet 2 of Autumn.bmp
⑥ Transmit packet 2 of hamlet.txt
(b)

Fig. 6.8 Explaining circuit switching versus packet switching. (**a**) Circuit switching, (**b**) Packet switching

$$T_2 = 0.182 \times 8/10 = 0.146 \text{ s}.$$

However, it starts at 0 s and finishes at $T_1 + T_2 = 7.46$ s.
The third task to download ucas.bmp needs

$$T_3 = 0.81 \times 8/10 = 0.648 \text{ s}.$$

The third task starts at 0 s and finishes at $T_1 + T_2 + T_3 = 8.11$ s.

With circuit switching, a dedicated circuit is established between the two parties of a communication session. Only at the end of the session is another dedicated circuit established (the circuit is "switched") to serve another session.

Circuit switching has the advantage of guaranteeing the quality of service of a communication session, which is not interfered by other communication sessions. This is especially important for real-time services such as telephone voice communication, where interference may cause voice sound quality degradation. Circuit switching also has disadvantages. Dedicated use of a channel implies that the channel capacity is not necessarily fully utilized. Furthermore, a communication session forces other sessions to wait till it finishes.

With packet switching, each of the three files (messages) is divided into a number of packets (e.g., each of 1 KB), and the channel statistically mixes and transmits the

Table 6.2 The Ethernet packet format, where 42-1500 bytes are used for the payload

7	1	6	6	2	**42-1500**	4
Preamble	Frame Delimiter	Destination MAC Address	Source MAC Address	Type	**Data (Payload)**	CRC

Table 6.3 Parts of the Web over Internet protocol stack

Layer	Protocol	Purpose
Application Layer (Layer 5)	HTTP	Access hypertext resources on a Web server from a Web client
Transport Layer (Layer 4)	TCP	Reliably transfer packets between two Internet hosts
Network Layer (Layer 3)	IP	Transfer packets between two Internet hosts in the best-effort way
Data Link Layer (Layer 2)	Ethernet, WiFi	Reliably transfer packets between two homogeneously connected devices
Physical Layer (Layer 1)	Wired or wireless, electrical or optical, cables or waveforms	Provide physical communication channels Transfer signals of individual bits

packets of all three messages, as illustrated in Fig. 6.8b. It is left as an exercise to show that the three downloading tasks starting at time 0 finish at $T_1 = 8.11$ s, $T_2 = 0.44$ s, and $T_3 = 1.44$ s, respectively. The three communication tasks proceed simultaneously, without waiting for one another to finish.

The following Table 6.2 shows the format of an Ethernet packet. Each packet has a **body** and a **header**. The body (in bold) is the payload data of the packet. The header, parts of which may come after the payload data, includes control information, addresses, and Cyclic Redundancy Check (CRC) error check information.

Another basic innovation of the Internet is to use multiple layers of protocols, stacked on top of one another, to form the widely used **TCP/IP protocol stack**. Each layer serves a particular purpose of networking by transferring different types of packets in different ways. Together, the stack of multiple layers of protocols effectively supports the global Internet we have today.

This idea was extended later by a Web layer on top of the Internet. This HTTP protocol enables accessing of hyperlinked hypermedia resources, such as webpages. Table 6.3 shows parts of the Web over the Internet protocol stack. Let us elaborate this five-layer stack from bottom up.

- Layer 1 is called the **physical layer**. It provides physical communication channels by wired electrical cables or optical fibers, as well as wireless waveforms. Layer 1 works on physical signals of individual bits of 0's and 1's.
- Layer 2 is called the **data link layer**. It reliably sends a layer-2 packet, called a *frame*, between two homogeneously connected devices, including hosts and networking devices. Two widely used protocols are Ethernet (as specified by the IEEE 802.3 standard) and WiFi (as specified by the IEEE 802.11 standard).

- Layer 3 is called the **network layer**, which has only a single protocol: IP (the Internet Protocol). It sends a layer-3 packet, called a *datagram*, between two Internet hosts, without guaranteeing reliable packet delivery. This is called the *best-effort* approach. The two Internet hosts are not necessarily homogeneously connected, but may be connected through one or more routers.
- Layer 4 is called the **transport layer**. A widely used protocol at this layer is TCP (the Transmission Control Protocol). It reliably sends a layer-4 packet, called a *TCP packet*, between two Internet hosts.
- Layer 5 is called the **application layer**. There may be several application layers stacked on top of one another. We only consider the Web application layer using HTTP (the Hypertext Transfer Protocol). Its purpose is to easily access hyperlinked hypertext resources, such as webpages, across the Internet.

6.3.1.1 How Does the Protocol Stack Work? Fetch a Home Page from a Server

Let us return to the example of Zhang Lei's using her laptop computer in Beijing to access a server in Shanghai, via the Web over Internet protocol stack.

In her Web browser, Zhang enters the following information to access the server: http://www.shanghaitech.edu.cn/ which has three fields. HTTP is the protocol name. This protocol is used to transfer Web requests and responses. The domain name www.shanghaitech.edu.cn identifies the website of ShanghaiTech, which is translated by DNS to the IP address 119.78.254.67 of the website server. The third part is a slash, indicating the **home page**, i.e., the introductory webpage of the website.

Upon receiving this HTTP request, the Web server at 119.78.254.67 sends back an HTTP response message, i.e., the contents of the home page. The message is divided into a number of HTTP packets.

In Fig. 6.9, an HTTP packet is indicated as a pink box, which is handed over to the TCP layer as the TCP packet body. The TCP layer adds a TCP header (shown as a blue box) to form a TCP packet. The TCP packet is handed over to the IP layer as the IP packet body. The IP layer adds an IP header (shown as a yellow box) to form an IP packet. Finally, the IP packet is handed over to the data link (Ethernet) layer as the packet body. The Ethernet layer adds an Ethernet header (shown as a red box) to form an Ethernet packet. We ignore the physical layer in this book.

Note that all these operations are automatically done in the server host. When a data link packet arrives at the destination host, i.e., Zhang's laptop computer, a reverse process takes place to recover the HTTP packet data.

Let us ask and answer several questions to better understand the protocol stack approach with packet switching communication.

- Does Zhang need to worry about TCP/IP and Ethernet when surfing the Web?
  The answer is NO. This is the beauty of the protocol stack approach.

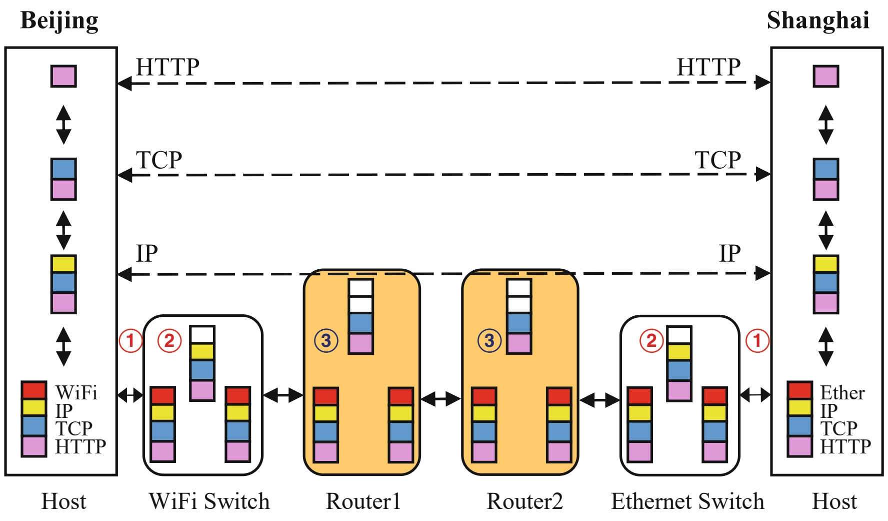


Fig. 6.9 Illustrating how a packet traverses the Web over Internet stack

A *user* at one layer does not have to worry about the layers below. The protocol stack provides two types of interfaces. When Zhang's Web browser in Beijing communicates with the Web server in Shanghai, both parties use a **peering interface** via the HTTP protocol, shown by horizontal dashed lines in Fig. 6.9. Here, *peering* means that the Web browser in Beijing acts as if it directly talks to its peer, the Web server in Shanghai, using a common HTTP protocol.

A software *developer*, however, implements the HTTP protocol using a **service interface** provided by the TCP layer, shown by vertical solid lines in Fig. 6.9. The TCP layer provides services to the HTTP layer via the service interface.

- Can the Web server in Shanghai send an HTTP packet to Zhang's Web browser in Beijing, without also sending a data link layer packet, e.g., an Ethernet frame?
  The answer is NO.

One cannot send a high layer packet without also sending a packet of every layer below. When a packet enters a network, it is in a data link layer format and travels as wired and wireless signals.

An HTTP packet must be repackaged as a TCP packet, which is then repackaged as an IP packet, which is then repackaged as an Ethernet packet, before it can be sent to the network as physical signals. This is similar to how post offices send letters. Letters are wrapped in envelopes, adding header information such as addresses, whether the letters should be express mailed, etc. The letters in envelopes are then wrapped in post office's packages, and so on.

- Can the server computer in Shanghai send an Ethernet packet to Zhang's personal computer in Beijing?

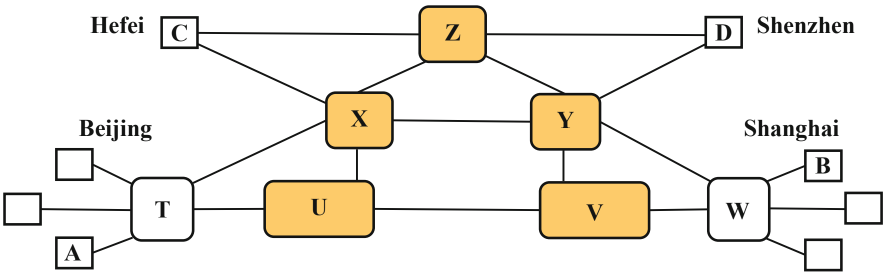


Fig. 6.10 Illustrating redundancy of the Internet: there are multiple paths from host A to host B

The answer is YES, as long as there is a homogeneous network connecting the two hosts via the same Ethernet protocol.

But the reality is like Fig. 6.9, to accommodate heterogeneity of the Internet. The server host sends an HTTP packet, wrapped as an Ethernet packet, to the Ethernet switch (①). The switch opens the packet to reveal the Ethernet header (shown as a white box in Fig. 6.9), and then adds a new header (with a new MAC address) to form a new Ethernet packet (②). When the packet arrives at Router2, the router opens the packet to reveal both the Ethernet and the IP headers (the two white boxes), and then form a new Ethernet packet by reformatting the packet and adding a new Ethernet header (③). Similar steps take place at Router1 (③) and the WiFi Switch (②), and then a WiFi packet arrives at the laptop computer host (①). Note that when Router1 (③) reformats the packet, it converts an Ethernet packet into a WiFi packet by adding a WiFi header.

- What is actually sent over a network?
  Bit strings of 0's and 1's.

Any packet is eventually encapsulated as one or more physical layer packets, which travel as wired or wireless signals. A physical layer packet is sent through electrical cables, electromagnetic waveforms or optical fibers, in a bit string of 0's and 1's. A 0 may be represented as a LOW voltage pulse or a LIGHTOFF state, while a 1 may be represented as a HIGH voltage pulse or a LIGHTON state.

- When a message is sent from a source host computer A to a destination host computer B, do all packets of the message travel through the same physical path from host A to host B?
  Not necessarily.

The global Internet has **redundancy** built in, to cope with traffic congestions and faults. This redundancy is a reason why we see many fault events reported every day worldwide, affecting parts of the Internet, but the global Internet as a whole has never gone down.

An example is shown in Fig. 6.10, where square boxes such as A, B, C, and D denote hosts, and the remaining nodes denote networking devices (switches and routers). There are multiple physical paths from host A to host B. When host A sends a 99-packet message to host B, it may well be the case that the first packet travels along the physical path A-T-X-Y-W-B, the 49th packet traverses path A-T-U-V-W-B, the 99th packet traverses path A-T-X-Z-Y-W-B, etc.

It may happen that the 49th packet arrives at host B before the first packet. Each packet comes with a packet number. The complete message is reassembled from the packets by their numbers, when all packets arrive at hots B.

As long as there is a valid path from A to B, a message can be communicated, even though other nodes or edges may be broken. For instance, when node T is down, host A cannot communicate. But, the remaining hosts B, C and D can still communicate with one another.

6.3.2 Elementary Web Programming

We introduce several Web programming examples to deepen the understanding of protocol stack. They also prepare students for the Personal Artifact project, which asks a student to create a **dynamic webpage**. Here, *dynamic* means that the contents or format of a webpage may change when displayed in a browser. In contrast, a static webpage does not change its contents or format.

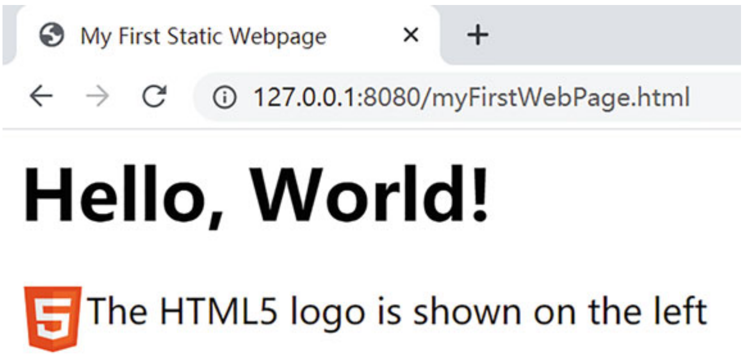
Only the basic HTML/CSS/JavaScript knowledge is introduced. Students are suggested to transfer the Go programming knowledge learnt to Web programming. They are also encouraged to learn any additional knowledge needed, by referencing the library of Personal Artifacts examples in the Supplementary Material.

6.3.2.1 Create a Web Server and Static Webpages

To create a Web server, add the following WebServer.go file in the working directory. The code uses the net/http package to create a Web server.

```
> cat WebServer.go
package main
import "net/http"
func main() {
 http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
        http.ServeFile(w, r, r.URL.Path[1:])
  })
  http.ListenAndServe(":8080", nil)
}
>
```

Then, run the program WebServer.go in background.



Hello, World!

The HTML5 logo is shown on the left

Fig. 6.11 Brower display of myFirstWebPage.html

```
> go run WebServer.go &
[1] 72
>
```

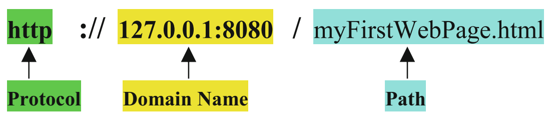
The '&' symbol at the end of a command indicates that the command runs in the background: Although control returns back to the shell, the WebServer program is still executing (with a process ID 72), and ready to accept Web requests.

As a second step, add the myFirstWebPage.html file to the working directory. This is probably the student's first webpage. The myFirstWebPage.html static webpage is explained in Example 6.3.

As a third step, open a Web browser and enter the following URL

http://127.0.0.1:8080/myFirstWebPage.html

Recall that a Web URL has three parts separated by "://" and "/". The domain name in this case is the loopback IP address of the student's laptop computer (the local host). We can also use the equivalent URL http://localhost:8080/myFirstWebPage.html. Port number 8080 is a default port for HTTP.



The above browser command will display a webpage shown in Fig. 6.11.

Example 6.3. Look Inside My First Webpage: myFirstWebPage.html
A webpage, as shown in Fig. 6.12, is expressed as Hypertext Markup Language (**HTML**) code, which is enclosed by a <html> . . . </html> pair and consists of a **head** and a **body**. The most recent version of HTML is HTML5.

```
<html>
   <head>
      <meta charset="utf-8">
      <title>My First Static Webpage</title>
   </head>
   <body>
      <h1> Hello, World! </h1>
      <p>
<img src="https://www.w3.org/html/logo/downloads/HTML5_Badge.svg"
style="float:left;width:30px;height:30px;">
The HTML5 logo is shown on the left
      </p>
      <script>
      </script>
   </body>
</html>
```

Fig. 6.12 A student's first webpage: myFirstWebPage.html

In the head part, <meta charset="utf-8"> specifies the character set as UTF-8, and the <title>…</title> pair specifies the title of the myFirstWebPage.html document as "My First Static Webpage". The title does not show up in the webpage itself when the HTML document is displayed.

There are three types of code in the body part. It pays for the lecturer to encourage students to play with the first webpage by changing different elements and items and then redisplay, to quickly appreciate the HTML and CSS code.

- Pure HTML code. One example is the <h1> … </h1> pair which displays the enclosed content "Hello, World!" as level-1 heading. Another example is a *paragraph* enclosed in a <p> … </p> pair. There is only one paragraph in the webpage myFirstWebPage.html. It contains two items: an HTML5 logo image specified by a <img …> pair, and a text phrase "The HTML5 logo is shown on the left". The src="…" item specifies the source URL (hyperlink) of the image. The HTML5 logo image is retrieved from the specified source URL (hyperlink) https://www.w3.org/html/logo/downloads/HTML5_Badge.svg. The protocol is "https", the domain name is "www.w3.org", and the local path name for the image file is "html/logo/downloads/HTML5_Badge.svg".
- Cascading style sheets (**CSS**) code specifying the format (presentation style) of the webpage. For instance, the item style="float:left;width:30px;height:30px;" in the paragraph says that the HTML5 logo image is displayed with the stated style: floating to the left of the text phrase, 30-pixel wide, and 30-pixel high.
- JavaScript code enclosed between a pair of <script> and </script>. In Fig. 6.12, this JavaScript part happens to be empty.

A Web server hosts the webpage files (called *resources*). The Web browser retrieves a Web resource, such as myFirstWebPage.html, to a student's laptop computer. The browser interprets it and displays the result. This is also called *rendering* the webpage. In this process, the browser automatically retrieves additional resources such as the HTML5 logo image by the hyperlink https://www.w3.org/html/logo/downloads/HTML5_Badge.svg.

Example 6.4. A Static Webpage Displaying the Date of Next Children's Day
We start to use JavaScript code in staticChildrensDay.html, as shown in Fig. 6.13. JavaScript is an *object-oriented* language. An **object** is a data structure with one or more components, each of which can be specified by the dot notation. An object may also contain a number of *methods* that operate on the data structure. Each method call is also specified by the dot notation. For instance, in the statement

```
document.getElementById("childrensDay");
```

document is an object, getElementById is a method, "childrensDay" is a parameter denoting the paragraph's ID. The statement says to get the element by ID in the HTML document, where ID is specified by the string "childrensDay".

The example code contains a single *paragraph* element. Its behavior is not directly specified in the <p>...</p> pair, but rather via the id "childrensDay". The <p>...</p> pair only specifies the location of the paragraph element on the webpage, but does not specify the content of the paragraph. Four lines of JavaScript code are included in the pair <script> ... </script>. They specify the style and the content of the paragraph element.

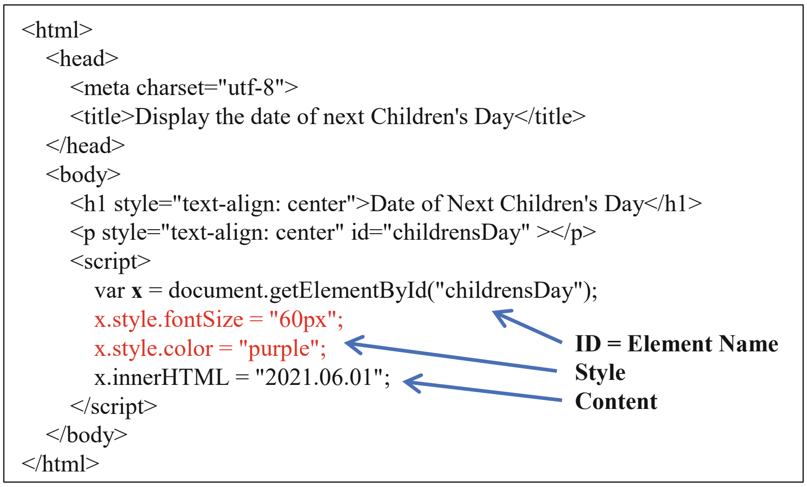
- var x = document.getElementById("childrensDay"); declares an *object* x and initializes it with the paragraph element denoted by ID childrensDay. Hereafter, x is the paragraph with id "childrensDay".
- x.style.fontSize = "60px"; sets the font size of paragraph x to be 60 pixels.
- x.style.color = "purple"; sets the color of paragraph x to be purple.
- x.innerHTML = "2021.06.01"; sets the content of paragraph x to be "2021.06.01".

6.3.2.2 Create a Dynamic Webpage

The previous example staticChildrensDay.html generates wrong outputs after June 1, 2021. To fix this error, we use the following JavaScript code to dynamically check and output the correct date, as shown in Fig. 6.14.

The program's logic is simple: (1) get the current values of year and month using the system provided Date object; (2) if the month is June or later, increments year.

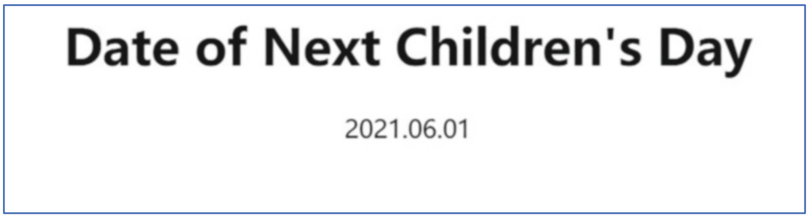
We add 1 to date.getMonth() because JavaScript counts months from 0 to 11.



```
<html>
   <head>
      <meta charset="utf-8">
      <title>Display the date of next Children's Day</title>
   </head>
   <body>
      <h1 style="text-align: center">Date of Next Children's Day</h1>
      <p style="text-align: center" id="childrensDay" ></p>
      <script>
         var x = document.getElementById("childrensDay");
         x.style.fontSize = "60px";
         x.style.color = "purple";
         x.innerHTML = "2021.06.01";
      </script>
   </body>
</html>
```

ID = Element Name
Style
Content

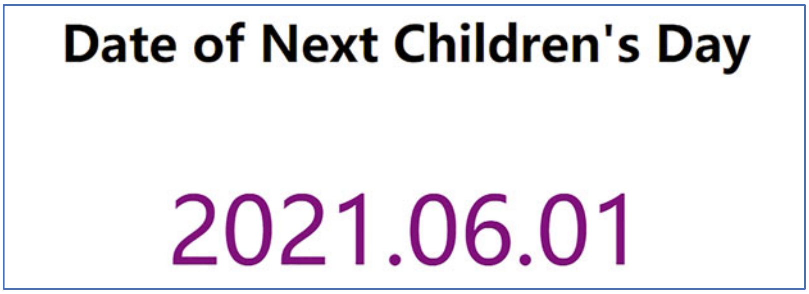
(a)



Date of Next Children's Day

2021.06.01

(b)

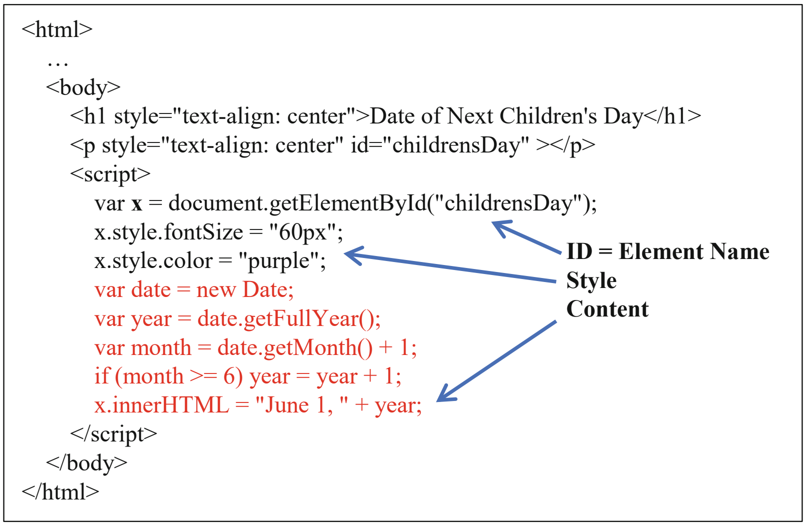


Date of Next Children's Day

2021.06.01

(c)

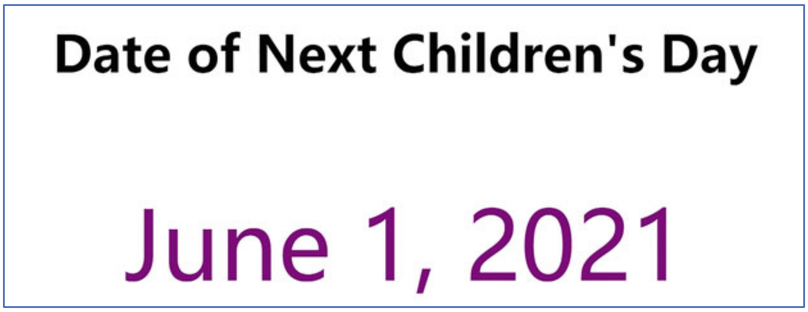
Fig. 6.13 A static webpage staticChildrensDay.html: code and output. (**a**) Sample code for displaying the date of next Children’s Day. (**b**) Output when the two lines of red code are deleted. (**c**) Output when the two lines of red code are present



```
<html>
  …
  <body>
    <h1 style="text-align: center">Date of Next Children's Day</h1>
    <p style="text-align: center" id="childrensDay" ></p>
    <script>
      var x = document.getElementById("childrensDay");
      x.style.fontSize = "60px";
      x.style.color = "purple";
      var date = new Date;
      var year = date.getFullYear();
      var month = date.getMonth() + 1;
      if (month >= 6) year = year + 1;
      x.innerHTML = "June 1, " + year;
    </script>
  </body>
</html>
```

ID = Element Name
Style
Content

(a)



Date of Next Children's Day

June 1, 2021

(b)

Fig. 6.14 A dynamic webpage ChildrensDay.html: code and output. (**a**) Code to display the dynamic date of next Children’s Day. (**b**) Browser output of ChildrensDay.html

6.4 Network Laws and Responsible Computing

Computer networks, especially the Internet, have shown great impact on human society, including our work, life, and culture. During the past decades, notable phenomena and principles were observed, which continue to evolve and impact society. We discuss several such phenomena and principles, as well as practical guides to responsible computing.

6.4.1 Bandwidth, Latency, and User Experience

We want a computer network to have high bandwidth and low latency. **Bandwidth** is the bit rate by which messages are transmitted over a network, while **latency** is the time taken to transmit a message.

Exact meanings of bandwidth and latency vary. Two types are often used.

- *Minimal latency and maximal bandwidth*. The total transmission time t of a message of length m bits transmitted between two parties can be estimated by the following **Hockney's formula**, $t = t_0 + m/r_\infty$. Here, t_0 is the minimal **latency**, i.e., the time needed to transmit a 0-bit message; and r_∞ is the maximal **bandwidth** achieved when the message length m approaches infinity.
- *User-experienced bandwidth and latency*. For a given message of length m, define the user-experienced latency as the total transmission time t and user-experienced bandwidth as the ratio m/t.

Recall *Keck's law* when we discuss the Wonder of Exponentiation in Sect. 1.3.2. Technology advances have enabled the exponential growth of the data transmission rate of optical fibers. The maximal bandwidth achieved in the so called hero experiments, roughly equal to r_∞, increased about 100 times every decade, as shown in Table 6.4. Note that t_0 and r_∞ represent extreme values. Latency t_0 here means the startup overhead of a message transmission, i.e., the minimal latency. Bandwidth r_∞ is is the maximal bandwidth.

Table 6.4 also shows the total transmission time t of a message of length m bits, assuming different values of t_0 and r_∞.

6.4.1.1 Bandwidth and Latency: Extreme and User-Experienced Values

Let us look at Table 6.4 again. Assume a message of length $m = 1$ GB $= 8$ Gb is transmitted on a network with startup latency $t_0 = 1$ ms and a maximal bandwidth $r_\infty = 10$ Tbps. The total transmission time is

Table 6.4 Bandwidth growth of a single optical fiber in hero experiments

Time of experiment	Maximal bandwidth achieved r_∞	Time t to Transmit 1 GB		Time t to Transmit 1 KB	
		$t_0 = 1\ \mu s$	$t_0 = 1\ ms$	$t_0 = 1\ \mu s$	$t_0 = 1\ ms$
1975	4.50E+07 bps, or 45 Mbps	178 s	178 s	0.2 ms	1.2 ms
1984	1.00E+09 bps, or 1 Gbps	8 s	8 s	9 μs	1 ms
1993	1.53E+11 bps, or 153 Gbps	52 ms	53 ms	1.1 μs	1 ms
2002	1.00E+13 bps, or 10 Tbps	0.8 ms	1.8 ms	1 μs	1 ms
2013	8.18E+14 bps, or 818 Tbps	11 μs	1 ms	1 μs	1 ms

We thank Mr. Jeff Hecht of IEEE to provide Donald Keck's original data

Table 6.5 Examples of user-experienced bandwidth and latency given $r_\infty = 1\text{Gbps}$

Minimal latency	$m = 1$ GB		$m = 1$ MB		$m = 1$ KB	
t_0	t	m/t	t	m/t	t	m/t
0.000001 s	8 s	1 Gbps	8 ms	1 Gbps	9 µs	0.89 Gbps
0.001 s	8 s	1 Gbps	9 ms	0.89 Gbps	1 ms	7.9 Mbps
0.1 s	8.1 s	0.99 Gbps	0.108 s	74 Mbps	0.1 s	80 Kbps
1 s	9 s	0.89 Gbps	1.008 s	7.9 Mbps	1 s	8 Kbps

$$t = t_0 + m/r_\infty = 1\text{ ms} + 8\text{ Gb}/10\text{ Tbps} = 0.0018\text{ s} = 1.8\text{ ms}.$$

In contrast, the user-experienced latency is $t = 1.8$ ms, and the user-experienced bandwidth is $m/t = 8$ Gb/1.8 ms = 4.44 Tbps.

Let us look at Table 6.5, which shows a network closer to the reality experienced by an ordinary home user with a 1-Gbps optic fiber subscription. For short messages of $m = 1$ KB and a high startup latency $t_0 = 1$ s, the user actually gets only a bandwidth of 8 Kbps.

6.4.1.2 Data Compression: Lossless and Lossy Compressions

Data compression is the technique used to reduce the size of a file, such that a file can take less storage space and transmission time. **Lossless compression** can reduce the size of a file without losing information. **Lossy compression** could lose information when reducing the size of a file. With lossy compression, the original file cannot be recovered from a compressed file.

The Linux operating system comes with a lossless compression command called gzip. Try this compression command

```
> gzip Autumn.bmp
```

and compare Autumn.bmp to the compressed file Autumn.bmp.gz, to see how much the file size is reduced. The command gzip is lossless, in that no information is lost. We can execute a decompress command:

```
> gzip -d Autumn.bmp.gz
```

to recover the original file Autumn.bmp.

In general, when compressing a program file or a scientific data file, we should use lossless compression. This is due to the fact that a single faulty bit could invalidate the entire file. On the other hand, lossy compression can be used with multimedia such as image, video, and sound files. We can reduce file size while losing some resolution (quality) of the multimedia file.

6.4.2 *Network Effect*

Since the invention of ARPANET in the 1960s, the global Internet has grown significantly in its scale and impact to society. Some facts and projections are summarized in Table 6.6. The impact of a network can be beneficial, harmful, or both. When the impact is beneficial, we often call it the network value.

Network effect refers to the phenomenon that a node derives value from the network, not from itself. An isolated node, e.g., a standalone laptop computer, has no impact to and receives no impact from the network. But when the laptop computer joins the Internet, it obtains much more value. This additional value is the network effect, in contrast to the standalone value. Network effect has two notable special cases: *superlinear total* and *viral growth*.

- *Superlinear total*. The total impact of a network is more than the linear summation of the impacts of its nodes. In other words, with a network of n nodes, the total impact of the network to society is super-linear, i.e., more than order of n.
- *Viral growth*. The size of the network can grow quickly like a virus spreading.

6.4.2.1 Metcalfe's Law and Reed's Law

The superlinear total feature of the network effect has been studied by scholars in computer science and other fields. People proposed observations and viewpoints, and some of which became known as "laws". Two of the most popular are Metcalfe's law which postulates a quadratic relationship, and Reed's law which postulates an exponential relationship.

These two laws are summarized in Box 6.2. They have caused extensive discussions not only in the scientific and technology circles, but also in social sciences and business communities.

Table 6.6 The Internet's historic scale growth trend and exemplar techniques

Time	# Nodes	Exemplar Techniques
1960s	A few	Packet Switching Network
1970s	Thousands	TCP/IP, Ethernet, Router
1980s	100 thousand	Client-Server Computing
1990s	Million	World Wide Web
2000s	100 Million	Cloud Computing
2010s	Billions	Smartphones, Mobile Internet
2020–2050	Trillions	Internet of Human-Cyber-Physical Systems

Box 6.2. Network Effect Laws
Metcalfe's law ($V \propto n^2$) is an observation made in 1980 by Robert Metcalfe, an American engineer and the inventor of Ethernet. It says that the value V of a network of n nodes is proportional to n^2, that is, the effect of the network is proportional to the square of the number of nodes of the networked system.

Reed's law ($V \propto 2^n$) is an observation made in 2001 by David Reed, an American computer scientist and the designer of the User Datagram Protocol (UDP) of the Internet protocol stack. Reed's law says that the value V of a network of n nodes can scale exponentially with n, because the network can form 2^n subgroups.

A common criticism to these laws is that they lack support from real-case data, although they may have captured some intuitive truth.

Recently, researchers used real data from two companies to validate the network effect laws. The results show that Facebook and Tencent data indeed validate Metcalfe's law.

Let us make two assumptions: (1) the number of nodes, n, is measured by the Monthly Active Users (MAUs); and (2) the value of a network is measured by the annual revenue of the company. Then, the revenue and MAU data of Facebook and Tencent, from the year 2003 to the year 2019, fit the following equations:

$$\text{Facebook}'\text{s Value} = 9.69 \times 10^{-9} \times n^2 \text{ USD}, \text{RMSD} = 4.58 \text{ Billion USD}$$

$$\text{Tencent}'\text{s Value} = 9.67 \times 10^{-9} \times n^2 \text{ USD}, \text{RMSD} = 4.30 \text{ Billion USD}$$

In other words, real data do show that $V \propto n^2$. RMSD is an acronym for the Root Mean Square Deviation, a fitting error metric. They are in the range of 4.30–4.58 billion US dollars, small numbers comparing to Facebook's revenue of 70.7 billion US dollars in 2019.

In fact, real data fit a "cube law" even better, with smaller RMSD.
$\text{Facebook}'\text{s Value} = 4.44 \times 10^{-18} \times n^3$ USD, RMSD=0.81 Billion USD

$$\text{Tencent}'\text{s Value} = 5.73 \times 10^{-18} \times n^3 \text{ USD}, \text{RMSD} = 3.38 \text{ Billion USD}$$

The value of a network falls somewhere between Metcalfe's law and Reed's law.

It is also interesting to note that the per-user revenue numbers of Facebook and Tencent follow a similar growth trend, shown in Fig. 6.15. In 2018, each user contributed about 24 US dollars to a company's revenue.

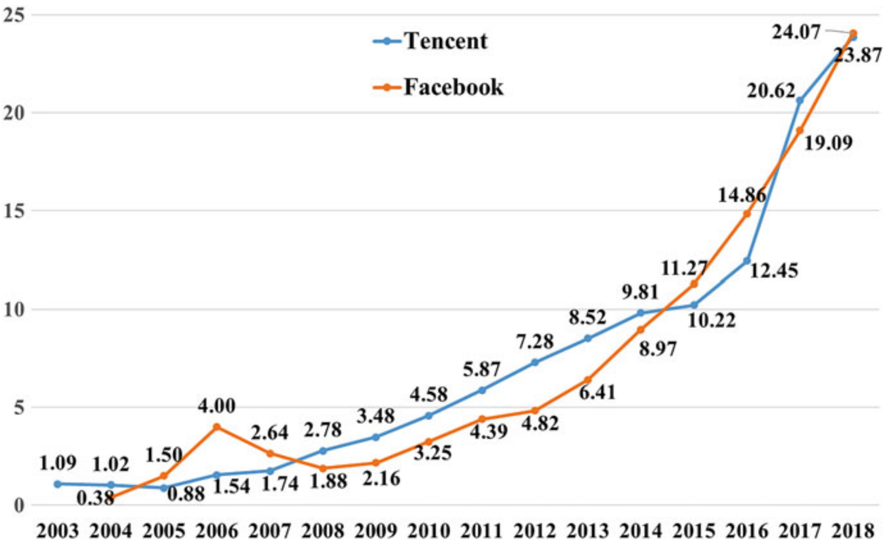


Fig. 6.15 Per-user revenue (amount in USD) growth trends of Facebook and Tencent

6.4.2.2 The Viral Marketing Phenomenon

Viral marketing refers to the following phenomena and practices: (1) viruses, including biologic viruses and computer viruses, spread wide and fast; (2) people learn from how a virus spreads to grow the market of a desirable computing product or service; (3) some computing products or services do grow their markets wide and fast, often enhanced by the network effect. When the market of a product or service grows wide and fast, we say it *becomes viral*.

The term *viral marketing* has become popular since 1997, when an email Web service called Hotmail quickly grew its user base to pass 10 million. Viral marketing has since become a marketing strategy in the field of business management. People have summarized four viral marketing features to characterize why computer viruses spread so wide and fast. A computer virus is (1) connected to the Internet, and has (2) zero purchasing price, (3) zero usage cost, and (4) zero propagation cost.

- *Connected*. This is obvious. Isolated viruses cannot infect.
- *Zero purchasing price*. No one pays any money to buy a computer virus. The virus comes “free of charge”. Similarly, the Hotmail service is free of charge.
- *Easy to use, zero usage fee*. One does not need to learn any manual to “use” a computer virus. “Using” a virus is also free of charge. Similarly, Hotmail is easy to subscribe and use.
- *Easy to propagate, zero propagation fee*. When a product is adopted by a user, there is an easy way for the product to propagate to other users. For instance, the Hotmail company devised a viral marketing trick: at the bottom of every e-mail

sent out by any Hotmail user, there is a line saying “Get your free e-mail at Hotmail”. This line prompts a non-user to sign up as a Hotmail user.

Example 6.5. Social Networks Facilitate Viral Marketing
Social networks encourage free sharing of information, through various sharing tools such as blog followers, Twitter tags, WeChat moments, and TikTok likes. Not only can a network grow quickly, so can a subnetwork. An individual grassroot user can grow her followers substantially, to become an important influencer in a short time.

For instance, “李子柒 Liziqi”, a rural youth based in Sichuan, China and a “Nature and Internet Celebrity”, has over 13 million followers on YouTube, as of December 9th, 2020, with half a million new subscribers in a month. Her videos of rural beauty have not only entertained people, but also sparked academic studies in network influencers, digital divide, and urbanization.

6.4.3 Responsible Computing

Responsible computing refers to the ideas and practices to design and use computing products and services responsibly. Responsible computing issues include, among other things, (1) cybersecurity issues, (2) privacy awareness, and (3) respecting professional norms.

6.4.3.1 Cybersecurity Issues

An obvious fact is that some information on the Internet is credible and some is not. A not so obvious fact is that not all information one receives from a trusted source, such as a trusted website, is credible. A main reason why we cannot always enjoy trusted Internet and Web services is that the global Internet is constantly under attack. According to a 2020 cybercrime report by the cybersecurity firm McAfee, cybercrime costed companies worldwide over 1 trillion US dollars.

Cybersecurity problems involve hardware, software and people. They come in many forms, including the following:

- *People*. Humans make mistakes, including users, developers, and administrators. Such errors create vulnerabilities which are exploited by attackers. Computing innovations, including hardware, software and services, are created by people. Such an innovation, e.g., viral marketing, can have both beneficial and harmful effects, even for the same person.
- *Malware*. Various malicious software enables an attacker to damage or gain unauthorized access to a computer. Malware examples include computer viruses, Trojan horses and spyware outlined in Box 6.3.
- *Hardware exploitation*. In 2018, researchers discovered an attack technique called Meltdown that “does not rely on any software vulnerabilities.” Instead,

the attack exploits a feature of processor hardware called out-of-order execution, to enable an attacker to read privileged information such as passwords.

- *DoS attacks*. A **denial-of-service** attack overwhelms a computer by sending it a lot of messages in a short amount of time, so that the computer has no resource left to handle normal requests. A **distributed denial-of-service** (DDoS) attack utilizes many computer hosts on the Internet to mount the attack.
- *Spams*. Unwanted emails are often called **spams**. Some spams are easy to identify, such as mass advertising emails and phishing emails. Other times, it is difficult to correctly classify an email as a spam. Suppose a student, who is a student member of both ACM and IEEE, receives a Call for Participation email from the Publicity Chair of an ACM or IEEE international conference. Should this be considered a spam? Professional sources usually offer an option for the student to indicate "I do not wish to receive such emails anymore".
- *Phishing*. A **phishing website** or **phishing emails** ask users for sensitive information, such as a person's password or credit card number. Similar to fishing, phishing offers various types of baits, such as "You are about to receive a subpoena. Click the attachment for details", or "Your account has been frozen and we need your PIN to unfreeze it".

Box 6.3. Computer Bugs, Malware, Viruses, and Trojan Horses

Computer **bugs** are the term used to refer to all errors in a computer or a network of computers, including software bugs and hardware bugs. This term was attributed to Grace Hopper, a computer pioneer who recorded in 1947 the first bug in computer history: a moth stuck in a relay in a Mark II Computer at Harvard University.

Computer bugs are unintentional errors. In contrast, **malware** is malicious software that is designed to intentionally damage or control a computer. Malware includes viruses, Trojan horses, spyware, etc. **Trojan horses**, like the one in the Greek story, are computer programs which hide their true intentions by disguising as ordinary computer files. **Spyware** collects information about an intruded computer and reports the information back to the attacker.

A **computer virus** is a self-replicating program able to infect computers, like biologic viruses infecting people. A computer virus needs to embed in another program (infecting the program). In contrast, a **computer worm** is a *standalone* self-replicating program that can spread to other computers.

Fred Cohen, an American computer scientist, demonstrated the first computer viruses in 1983 while he was a graduate student at the University of Southern California. He demonstrated that a virus-infected Unix command could let the virus gain control of a computer in 5 min.

Various cybersecurity techniques and processes have been utilized to counter cybercrimes. These counter measures have overheads and are not foolproof.

- *Physical isolation*. For instance, core computing systems in some financial institutions are not connected to the Internet.
- *Firewalls*. A computing system connected to the Internet can be protected by a **firewall** installed between the system and the Internet, to block or filter out undesirable messages. When an institution has multiple computers at multiple sites, they can be interconnected by virtual private networks (**VPNs**), which are secure networks built on top of the public Internet. Such an institution system, protected by firewalls and VPNs, offers an approximation to physical isolation, without the cost of building a dedicated system.
- *Antivirus software*. As the name implies, **antivirus software** is used to detect and kill computer viruses. In fact, modern antivirus tools can detect and remove various malware, including viruses, Trojan horses, and spyware.
- *Cryptography*. Many cybersecurity tools utilize results from **cryptography**, a professional field that studies secure message communication in the presence of adversaries. The basic idea is **encryption**: turn the original message (*plaintext*) into an encrypted message (*ciphertext*). **Decryption** is the reverse process of recovering the plaintext from the ciphertext.

Two types of cryptography are popular. They both use **keys** for encryption and decryption. A key is a specially designed piece of information. In **symmetric-key encryption**, the two parties of communication share the same secrete key for both encryption and decryption. The sender side computer uses an *encryption algorithm* to encrypts plaintext into ciphertext with the key. Then the ciphertext is sent out. The receiver side computer uses the same key and a *decryption algorithm* to decrypt the ciphertext received, to recover the original plaintext.

For instance, the **Caesar cipher** may use a value, e.g., 3, as the key. Its encryption algorithm shifts each plaintext letter 3 positions down the alphabet, and the decryption algorithm shifts each ciphertext letter 3 positions up. Thus, given the plaintext message "HELLO", the sender encrypts it to create and send out the ciphertext "KHOOR". Upon receiving the ciphertext, the receiver decrypts it to recover the original message "HELLO", using the same key (3).

In contrast to symmetric-key encryption, the encryption key and the decryption key are different in **public-key encryption**. Let us use a simplified example to demonstrate how public-key encryption works.

Example 6.6. Public-Key Encryption Using the RSA Method
Suppose a sender computer wants to send the receiver computer a message. The receiver publishes her encryption key for the world to know. That is why it is called the receiver's **public key**. However, the receiver holds her decryption key secrete (private). Thus, it is called the receiver's **private key**. Note that only the *receiver's* keys are involved.

We use below an example from the original paper of the most popular public-key encryption method, known as **the RSA method**. Here, RSA refers to Ron Rivest, Adi Shamir, and Leonard Adleman, who are now professors at the Massachusetts Institute of Technology, University of Southern California, and the Weizmann

Institute of Science, respectively. They invented the RSA method when they were at MIT, and received Turing Award in 2002.

1. As a preparation, first encode a message into a sequence of numbers. Then it suffices to show how to securely communicate one number.
2. The sender uses his encryption algorithm F to generate the ciphertext. F takes the plaintext number M and the *receiver's* public key K_P as inputs, to generate the ciphertext number C. In other words, $C = F(M, K_P)$.
3. The ciphertext number C is transmitted from the sender to the receiver, over the Internet. An eavesdropper may intercept C. However, C is enciphered and does not reveal the plaintext M.
4. The receiver uses her private key K_S and the decryption algorithm G to convert the ciphertext number C into the original plaintext M. That is, $M = G(C, K_S)$.

Suppose the plaintext is a 20-character message "ITS ALL GREEK TO ME " (note that there are five space characters). The process goes as follows.

- Each character of the plaintext is first turned into a 2-digit number, by the following converting scheme: space=00, A=01, B=02, …, Z=26. The 20-character plaintext "ITS ALL GREEK TO ME " will result in a large number of 40-digit length: 0920190001121200071805051100201500130500, which is broken into ten 4-digit numbers: 0920 1900 0112 1200 0718 0505 1100 2015 0013 0500. We only need to consider how to securely communicate one number, e.g., 0920.
- The sender uses the following encryption algorithm: $C = M^e \bmod n$, where M is the plaintext number, C is the ciphertext number, and the pair (e, n) is the public key K_P. In other words, $K_P = (e, n)$. Now let us make *the critical magic assumption*: $n = 2773$, $d = 157$, $e = 17$. Then, when the plaintext number is $M = 0920 = 920$, the corresponding ciphertext number is easily computed:

$$C = M^e \bmod n = 920^{17} \bmod 2773 = 948 = 0948.$$

- The ciphertext number 0948 is transmitted from the sender to the receiver.
- The receiver uses the following decryption algorithm: $M = C^d \bmod n$, where M is the plaintext number, C is the ciphertext number, and the pair (d, n) is the private key K_S. With the ciphertext number 0948, the corresponding plaintext number is again easily computed:

$$M = C^d \bmod n = 948^{157} \bmod 2773 = 920 = 0920.$$

The entire message is a 20-character plaintext "ITS ALL GREEK TO ME ", which is converted into ten numbers: 0920 1900 0112 1200 0718 0505 1100 2015 0013 0500. These numbers are encrypted as 0948 2342 1084 1444 2663 2390 0778 0774 0219 1655. This encrypted number sequence is transmitted over the Internet.

Note that an eavesdropper knows many things about the communication, because all this information is public, including:

- The encryption algorithm F, which is $C = M^e \bmod n$, and the decryption algorithm G, which is $M = C^d \bmod n$.
- The public key $K_P = (e, n) = (17, 2773)$.
- The character-to-number converting scheme: space=00, A=01, B=02, ..., Z=26.
- The ciphertext number sequence 0948 2342 1084 1444 2663 2390 0778 0774 0219 1655, which is transmitted over the open Internet.

The eavesdropper can use the character-to-number converting scheme space=00, A=01, B=02, ..., Z=26, to try to recover the plaintext number sequence, which yields the following gibberish message:

I?W?J?N?Z?W?G?G?BSP?, (symbol '?' is for an undefined number)

instead of the plaintext message:

ITS ALL GREEK TO ME .

Why cannot the eavesdropper decode the ciphertext? He lacks the private key $K_P = (d, n) = (157, 2773)$, which only the receiver holds.

Let us return to the *magic assumption*: $n = 2773$, $d = 157$, $e = 17$. How are these values determined? There are profound mathematical insights underlying this assumption. We go through the basic idea without involving mathematic details. The receiver decides the values as follows.

- Randomly choose two large prime numbers p and q, and set $n = p \times q$. For this example, the receiver chooses $n = p \times q = 47 \times 59 = 2773$.
- Compute the Euler number $(p - 1) \times (q - 1) = 46 \times 58 = 2668$.
- Randomly choose a large integer d such that $\text{GCD}(d, 2668) = 1$. For this example, she chooses $d = 157$ which satisfies $\text{GCD}(157, 2668) = 1$. Now the receiver has the complete private key information: $K_S = (d, n) = (157, 2773)$.
- Find value e satisfying $(d \times e) \bmod 2668 = 1$. For this example, $e = 17$. Now the receiver can publish the public key: $K_P = (e, n) = (17, 2773)$.

Note that the eavesdropper knows $n = p \times q = 2773$. However, he does not know the two large prime numbers p and q. Consequently, he cannot compute the Euler number $(p - 1) \times (q - 1) = 46 \times 58 = 2668$, or the number $d = 157$.

One may wonder once we know n, if we can determine the two prime numbers p and q by a clever algorithm. This is called the **prime factorization problem**, which has no known efficient algorithm. *RSA relies on this fact.*

As of the year 2020, the largest RSA integer factored is RSA-250, which has 250 decimal digits. A French-US team accomplished the prime factorization task utilizing a network of parallel computers in Europe and the USA. The total computing resources used are roughly 2700 core-years.

A computer network or application can use symmetric-key encryption, public-key encryption, or both. Let us look at **HTTPS**, the secure version of HTTP, which is the most popular protocol for accessing websites today. According to W3Techs, a Web technology survey service, HTTPS is the default protocol for 68% of the top 10 million websites worldwide.

HTTPS uses an encryption layer called *Transport Layer Security* (**TLS**), for secure communication between a Web browser and a website. TLS, thus HTTPS, uses both symmetric-key encryption and public-key encryption techniques. For the long term, HTTPS (and TLS) uses public and private keys between a browser and a server. For the short term, e.g., for each HTTPS GET session, a onetime symmetric session key is automatically generated from the public and private keys, and used by the browser and the server for this session.

When a browser gets a public key from a website, say, ccf.org.cn, how does the browser know for sure that the website is indeed for China Computer Federation? HTTPS also uses **digital certificate**, a bit string issued by a trusted institution called **certificate authority**, to authenticate that a website is who it claims to be. When the website sends the public key to the browser, the website actually sends the digital certificate which encloses the public key. The browser interacts with the certificate authority to make sure that the certificate is indeed from the website of China Computer Federation. Then the browser can use the pubic key, contained in the certificate, to securely communicate with ccf.org.cn.

6.4.3.2 Privacy Awareness

Security and privacy are related. Some people consider privacy issues a subset of cybersecurity issues. To differentiate, security is about safeguarding a user's systems and data from attacks, while privacy emphasizes keeping a user's identity and personally identifiable information (PII) *private*.

Personal information is any information relates to a natural person's identity or personally identifiable information, but does not include anonymized personal information. Such information is broad, including personal names, ID numbers, personal photos or videos, website clicks records, voice signals collected by a smart speaker, financial records, medical conditions, and much more.

Students should be aware that privacy can be violated not only when one's obviously sensitive personal data, such as name and credit card number, are explicitly exposed. Technical and social means exist to reveal one's personal data, such as by utilizing metadata, data mining, and artificial intelligence techniques.

For instance, smart meter technology is used by some communities to care for senior citizens who have medical conditions and live alone. When the water usage pattern of a senior citizen's apartment falls below a certain threshold curve, it is likely that the senior citizen is incapacitated and a health worker should go to check up. This is especially beneficial in a pandemic, where more senior citizens are staying at home alone. However, such technology should not be misused.

Websites collect personal data to provide better services and personalized advertisements. Security cameras are installed in residential areas and city blocks to fight crimes. However, they also raise privacy concerns. Where and how to draw the line is still an on-going research area, in the computer science field as well as in the legal field. *IEEE Security and Privacy* is a professional magazine exploring security and privacy issues. On the legal side, the European Union enacted a law framework, called *General Data Protection Regulation* (**GDPR**), which went into effect in 2018. The Chinese law-making body, the National People's Congress of the People's Republic of China, published a request for comments of a Personal Information Protection Act Draft (**PIPA**), in October 2020.

We list below several items of the laws, in a non-legal language:

- The laws facilitate the protection, as well as legal and fair use, of personal information (personal data).
- A person has basic rights to his/her personal information, such as:
  - Right to permit a third party to collect and use personal data
  - Right to timely rectification of personal data
  - Right to be forgotten
  - Right to port one's personal data to another website
- These rights are protected by law, even when a piece of personal data is not owned by the person. A person's cellphone number is protected, even though the number belongs to the telecom company, and the person only "rents" it.
- Another person or institution can collect, store, process, and otherwise use a person's data in a legal and fair way (PIPA used 合法, 正当, 必要).

6.4.3.3 Respecting Professional Norms

Computing innovations have **beneficial and harmful impact** on society. The even more challenging part is that the same computing innovation could be both beneficial and harmful, sometimes even to the same person. For instance, the personalized recommendation algorithm used in a video website can recommend relevant videos to a targeted user, which is often beneficial. On the other hand, the same algorithm can create a somewhat closed, even biased, information world to the same user, which may be harmful.

Another challenge relates to **collaboration**. A modern worker seldom works in complete isolation. The Human Sorter project of this book specifically asks students to form groups to do team work. Computing innovations include collaboration tools, such as emails, social network groups, document-sharing software, video conference software, and peer-programming tools. Collaboration often increases productivity and result quality, by synergizing experiences, perspectives, talents, and skills from the team members. However, collaboration without respecting professional norms could decrease productivity or even cause harm. Sometimes, the line of right and wrong is not clear, requiring thoughtful judgement.

One way to deal with these dilemmas is to obtain help from professional communities. As users and developers, students need to be aware of and respect professional norms, which are often codified in a professional society's bylaws. For instance, the Association for Computing Machinery maintains Bylaw 15: *ACM Code of Ethics and Professional Conduct*. Its seven principles are shown in Box 6.4.

Box 6.4. Seven Principles of the ACM Code of Conduct

1. Contribute to society and to human well-being, acknowledging that all people are stakeholders in computing.
2. Avoid harm.
3. Be honest and trustworthy.
4. Be fair and take action not to discriminate.
5. Respect the work required to produce new ideas, inventions, creative works, and computing artifacts.
6. Respect privacy.
7. Honor confidentiality.

Let us look at Principle 5 in more details. In plain language, this principle says that the creator and the user of a piece of intellectual work should respect each other. A new computer innovation, with big or small impact, is built using prior work. A creator is also a user. Respect comes in many forms, including the following:

- Acknowledgement. The user should credit and cite prior work used.
- Respecting various protections. The user should respect not only laws, but also regulations, non-disclosure agreement, and code of conduct, which protect the intellectual property rights of the creator.
- Contribution to public good. The creator should not oppose fair use of his work, and should contribute to open source or public domain work. The world's most popular operating system is Linux, not any proprietary one.

Whether a conduct is professional (i.e., right or wrong in plain language) can sometimes be difficult to determine. Thus, not only awareness, but also thoughtful personal judgement, are needed. We use three examples to illustrate the difficulty.

Example 6.7. Free Flow Versus Professionally Sharing of Scientific Data
Should scientific data flow freely, or in some constrained way?

It seems obvious that scientific data should flow freely. For instance, genome sequence data of COVID-19 viruses are freely available at three databases hosted in USA, Germany and China: GenBank, GISAID, and 2019nCoVR. Anyone can go to these websites and download a genome sequence data file. This helps scientists from all over the world to fight against the COVID-19 pandemic.

However, scientific data may contain sensitive information. Simple-minded insistence on free flow of scientific data may generate privacy, confidentiality, safety, and security concerns. A better practice is to share scientific data following professional norms, which address these concerns.

In fact, this point was made explicit by those scientists who create the GISAID initiative. Look at https://www.gisaid.org/ to find out how they created a model of free access, while "overcoming disincentive hurdles and restrictions", by requiring users to identify themselves and uphold the GISAID Database Access Agreement.

Example 6.8 Full Disclosure Versus Responsible Disclosure
Suppose a worker in a company finds a vulnerability in a product of the company. The product is used by billions of users. Hackers could exploit the vulnerability to cause harm to society. What should the worker do? Two of the choices are listed below:

- *Responsible disclosure*. Report the vulnerability to the company, without disclosure to the public. This gives the company time to fix the vulnerability, without hackers knowing and exploiting it.
- *Full disclosure*. Announce the vulnerability to the public. The public pressure will force the company to fix the bug before hackers can exploit it.

It is left as an exercise for students to decide which choice better matches the spirit of the ACM Code of Conduct.

Example 6.9 Morris Worm
In 1988, Robert T. Morris, then a first-year graduate student in computer science at Cornell University, released a computer worm (later called **Morris worm**) to the Internet, while doing his research work. Morris intended to count the number of computers on the Internet. He did his work by utilizing unauthorized accesses to spread the worm on those computers.

Due to a bug in the program, the worm caused damages in millions of dollars on thousands of computers. Morris was sentenced to 3 years of probation plus community work and a fine. Later, Robert T. Morris earned a Ph.D. degree from Harvard University, became a professor at MIT, and was elected to the US National Academy of Engineering, for his contributions in the computer network.

Given the above facts, determine if releasing the Morris worm is wrong, according to the ACM Code of Conduct. This is left as an exercise.

6.5 Exercises

1. Which of the following statements is NOT true regarding network thinking?

   (a) A node in a network does not have to be a computer.
   (b) A node in a network is an abstract or real entity.
   (c) Nodes of a network do not have to send messages to one another.

   (d) There can be no isolated node in a network. Every node has to connect to at least one other node.

2. Which of the following statements is NOT true?
   (a) An accessing point (AP) is not a host, but a networking device, which converts wired and wireless signals to each other.
   (b) A network switch connects multiple devices running the same protocol, to form a homogeneous network.
   (c) A network switch can be used to connect a LAN running Ethernet and another LAN running WiFi, to form a heterogeneous network.
   (d) Several functions can be packed into a product. For instance, a WiFi device can combine the AP, switch, and router functions into the same the product, called a **WiFi router**.

3. What are the sizes of the following networks? Fill out the following table.

Namespace	Example	Network size (number of nodes)
Personal name	Joan Smith	Billions
WeChat user	ZhongguanVillager	
URL	www.ict.ac.cn/cs101	
Internet site	www.ict.ac.cn	
Email address	zxu@ict.ac.cn	
IP address	159.226.97.84	
Phone number	189-6666-8888	
MAC address	00-1E-C9-43-24-42	

4. Determine uniqueness and user-friendliness of each of the following naming schemes of networks. Fill out the blank cells in the table.

Namespace	Example	Uniqueness	User-friendliness
Personal name	Joan Smith	Not unique	Friendly
WeChat user	ZhongguanVillager		
URL	www.ict.ac.cn/cs101		
Internet site	www.ict.ac.cn		
Email address	zxu@ict.ac.cn		
IP address	159.226.97.84		
Phone number	189-6666-8888		
MAC address	00-1E-C9-43-24-42	Unique	Not user friendly

5. When accessing a website by entering https://www.ict.ac.cn/cs101 in a browser, what is the top-level domain (the highest level domain)?
   (a) https
   (b) www
   (c) cn
   (d) cs101

6. Which of the following is NOT a legitimate IP address when using IPv4?
   (a) 0.0.0.0
   (b) 127.0.0.1
   (c) 159.226.97.84
   (d) 159.279.97.84
7. Which of the following is a legitimate IP address when using IPv4?
   (a) 159.226.97.0
   (b) 389.226.97.84
   (c) 159.389.97.84
   (d) 159.279.389.84
   (e) 159.226.97.389
8. What is the role of the Domain Name Service (DNS)?
   (a) Converting an Internet domain name into an Internet Protocol address.
   (b) Converting an Internet Protocol address into an Internet domain name.
   (c) Converting an IP address into a domain name.
   (d) Converting a cellphone number into an Internet domain name.
9. IPv6 has a 128-bit address format. In contrast, IPv4 has only a 32-bit address format. More IP addresses can be provided by IPv6, compared to IPv4. But how many more?
   (a) 32 times as many IP address as IPv4.
   (b) 96 times as many IP address as IPv4.
   (c) 128 times as many IP address as IPv4.
   (d) 2^{32} times as many IP address as IPv4.
   (e) 2^{96} times as many IP address as IPv4.
   (f) 2^{128} times as many IP address as IPv4.
10. All scientific literature of the world forms a graph. Let us call this graph the *scientific literature graph* (SLG), where a paper (or a book) is a node (vertex), and a citation is an edge pointing from the citing work to the cited work. According to network thinking, is the SLG a network?
    (a) No. A computer network is used to pass messages. No message is communicated in the SLG. The citations are just marks.
    (b) Yes. The SLG depicts the connectivity of the network of scientific literature.
    (c) No. Network thinking must utilize both abstractions of connectivity and protocol stack.
    (d) No. The SLG is not network because scientific literature keeps growing.
11. According to network thinking, is the scientific literature graph (SLG) a static network, a dynamic network, or an evolving network?
    (a) The SLG is not a network.
    (b) The SLG is a static network.

(c) The SLG is a dynamic network.
(d) The SLG is an evolving network.

12. Albert Einstein published in 1905 a paper on special relativity, with the title *On the Electrodynamics of Moving Bodies* when translated into English. This paper contains no citation. Recall that in the SLG, a citation is an edge pointing from the citing work to the cited work. Which of the following statements is correct?

    (a) In the SLG, Einstein's paper is a node with no incoming edges.
    (b) In the SLG, Einstein's paper is a node with no outgoing edges.
    (c) In the SLG, Einstein's paper is an isolated node, with neither incoming nor outgoing edges.
    (d) Einstein was wrong not citing prior work.

13. The second-generation search engines generated much better results than the first-generation search engines. Why?

    (a) The second-generation search engines used much better computer systems.
    (b) The second-generation search engines collected user data, such as a user's click history data, to improve search results.
    (c) The second-generation search engines utilized artificial intelligence techniques and were smarter.
    (d) These first-generation search engines only utilized nodes of the network of webpages. The second-generation search engines practiced network thinking better by utilizing both nodes and edges.

14. Let us define a search network as follows: the nodes are the search engine and all search engine users, and there exists an edge between a user and the search engine. Which of the following statements is NOT correct?

    (a) At any moment, the search network is a star network.
    (b) At any moment, the search network is a static network.
    (c) At any moment, the search network is a dynamic network.
    (d) The search network is an evolving network.

15. Which of the following statements is NOT correct?

    (a) A bus is equivalent to a fully connected graph with at most one edge being active at any moment.
    (b) Suppose n nodes are connected by a crossbar switch with n fully-duplex ports. Such a network is equivalent to a fully connected graph of n nodes with at most n edges being active at any moment.
    (c) A network of n nodes connected by a crossbar switch is a dynamic network, because it does not change its nodes but may change its edges.
    (d) A network of n nodes connected by a crossbar switch is an evolving network, because it may change its nodes and edges.

16. Which of the following statements is correct regarding packet switching?

(a) It splits multiple users' multiple messages into packets, and statistically transmits the packets in turn.
(b) It splits only one user's multiple messages into packets, and statistically transmits the packets in turn.
(c) It packs one user's multiple messages into one packet and transmits it.
(d) It packs multiple users' multiple messages into one packet and transmits it.

17. Refer to Example 6.2. Verify that the three downloading tasks finish at T_1=8.11 s, T_2=0.44 s, and T_3=1.44 s, respectively.
18. Refer to Example 6.2. Observe that for both circuit switching and packet switching, all three downloading tasks finish at the moment of 8.11 s. In other words, packet switching does not save time. Then, why bother with inventing packet switching? Select all reasonable explanations.

    (a) With packet switching, the three communication tasks proceed simultaneously, without waiting for one another to finish.
    (b) With circuit switching, user Wang feels as if his computer is frozen. He needs to wait for 7.31 s before seeing any bits transmitted.
    (c) With circuit switching, user Zhang feels as if her computer is frozen. She needs to wait for 7.46 s before seeing any bits transmitted.
    (d) In many applications using circuit switching, e.g., two people having a telephone conversation over a circuit, the channel capacity of the circuit is often not fully utilized. Packet switching can more efficiently utilize the channel capacity, by having multiple communication tasks sharing the same circuit.

19. In general, a packet contains four types of information: payload data, addresses, control information, and error-handling information. What information is contained in the packet header and the packet body, respectively?

    (a) The packet header contains addresses, control information, and error-handling information; the packet body contains payload data.
    (b) The packet header contains addresses and control information; the packet body contains payload data as well as error-handling information.
    (c) The packet header contains addresses; the packet body contains payload data as well as control information and error-handling information.
    (d) The packet header contains control information and error-handling information; the packet body contains payload data and addresses.

20. Which of the following statements is NOT correct?

    (a) The Web over Internet protocol stack is a technical foundation of data communication for the World Wide Web.
    (b) The HTTP peering interface is used between two peers: a Web browser and a Web server. This peering interface provides an abstraction, such that the two peers do not need to worry about the layers of protocols below.
    (c) The service interface between HTTP and TCP is used for the TCP layer to support the HTTP layer.

   (d) All packets of an HTTP message from a browser to a server travels along the same physical path.

21. Which of the following statements is NOT correct?

   (a) When an HTTP packet is sent from a browser to a Web server, at least one TCP packet is also sent from the browser computer to the server computer.
   (b) When an HTTP packet is sent from a browser to a Web server, at least one IP packet is also sent from the browser computer to the server computer.
   (c) When an HTTP packet is sent from a browser to a Web server, at least one datalink layer packet is also sent from the browser computer to the server computer.
   (d) When an HTTP packet is sent from a browser to a Web server, a physical layer binary string of 0's and 1's is also sent from the browser computer to the server computer.
   (e) An HTTP packet can be sent from a browser to a Web server, without sending any TCP, IP, or datalink layer packets.

22. Refer to Fig. 6.10. Suppose only routers (shown as brown boxes) may become faulty. What is the minimal number of faulty routers required to disable communication between host A to host B?

   (a) 1
   (b) 2
   (c) 3
   (d) 4

23. Refer to Fig. 6.10. Suppose only inter-router edges may become faulty. What is the minimal number of faulty inter-router edges required to disable communication between host A to host B?

   (a) 1
   (b) 2
   (c) 3
   (d) 4

24. A student subscribes to an optical fiber plan from a reputable ISP, which connects his apartment to the Internet with a 1-Gbps bandwidth connection. However, he often only experiences a 5-Mbps bandwidth when accessing the Internet. Why is there this huge disparity? Which of the following is NOT a reasonable explanation?

   (a) The 1-Gbps bandwidth optic connection is only a portion of the full path from the student's laptop to the accessed website. The rest of the path could be much slower.
   (b) The student may be sharing a network switch with neighbors.

(c) Assume the rest of the Internet is fast enough and there is no sharing. The 1-Gbps bandwidth is the maximal bandwidth, i.e., r_∞ in Hockney's formula. User-experienced bandwidth can be much lower.
(d) The student has a fast laptop computer.

25. Which of the following statements is correct regarding data compression?

(a) Lossless compression often generates larger compressed files than lossy compression.
(b) Lossy compression can be used to compress a Go program file.
(c) Lossy compression can be used to compress a binary program file, i.e., an executable program file.
(d) To compress a picture file such as Autumn.bmp, one must use a lossless compression tool, such as the gzip command.

26. Which of the following statements is true about network effect?

(a) A laptop computer connected to the Internet has more value to the user than a standalone computer, because it benefits from network effect.
(b) The total value of a network is the linear sum of the values of its nodes.
(c) Reed's law is wrong because Facebook and Tencent data do not support it.
(d) Viral marketing is absolutely harmful, like biologic viruses hurting people.

27. Cybersecurity protection needs to consider (select one answer)

(a) Software
(b) Software and people
(c) Software and hardware
(d) Software, hardware, and people

28. Which of the following statements is true?

(a) Software bugs are a form of malware.
(b) Malware is a form of software bugs.
(c) Computer viruses are a form of malware.
(d) Spam is a form of software bugs.

29. Which of the following statements is true?

(a) A website has installed firewall, antivirus software and antispam software. It should be considered a trusted website.
(b) All information from a trusted website is credible.
(c) I access a website through HTTPS, the secure version of HTTP. Thus, the information I receives from the website is credible.
(d) There is neither perfect cybersecurity, nor absolutely trustworthy website.

30. I receive an email where the Subject part says a famous charity is asking for donation. Which of the following actions is proper?

(a) I should ignore the email since it is a phishing email.
(b) I should donate by clicking the URL in the email and fill out the form with my credit card information and donation amount.
(c) I should find out more details by clicking the attachment of the email.
(d) I should double check before the donation action.

31. Which of the following statements is true about the Caesar cipher?

(a) It uses public-key encryption.
(b) It uses symmetric-key encryption.
(c) It uses a combination of public-key and symmetric-key encryption.
(d) It uses no encryption.

32. Which of the following statements is true about HTTPS?

(a) It uses public-key encryption.
(b) It uses symmetric-key encryption.
(c) It uses a combination of public-key and symmetric-key encryption.
(d) It uses no encryption.

33. (***) Example 6.6 is mistaken. The eavesdropper CAN decode the ciphertext to get the plaintext. The eavesdropper knows the following public facts: (1) $n = p \times q = 47 \times 59 = 2773$; (2) $e = 17$; (3) the relation between the pair (d, e). He can use the relation $(d \times e) \bmod 2668 = 1$ to find d, and then generates the private key (d, e).
What is wrong with this reasoning?

(a) Fact (1) does not hold, because the eavesdropper does not know $n = 2773$.
(b) Fact (1) does not hold. The eavesdropper only knows $n = 2773$, but not $47 \times 59 = 2773$. Consequently, he cannot use $(d \times e) \bmod 2668 = 1$, because he cannot compute the Euler number $(p - 1) \times (q - 1) = 46 \times 58 = 2668$.
(c) Fact (2) does not hold, because $e = 17$ is not public knowledge.

34. (***) A student has found a bug in Example 6.6. The eavesdropper CAN decode the ciphertext by using a computer program to try all pairs of prime numbers smaller than $n = 2773$, to discover that the crucial hidden fact that $n = p \times q = 47 \times 59 = 2773$.
Suggest a way for the lecturer to fix this "bug"?

(a) Inform the class that this Example is from an authority, i.e., from RSA, the famous Turing Award winners.
(b) Inform the class that no such computer program exists, because prime factorization is a hard problem.
(c) Inform the class that Example 6.6 uses small numbers to illustrate the principle of the RSA method. Real applications use much larger n, much harder to break. For instance, breaking RSA-250 into two prime numbers needs 1000 years on a laptop computer. HTTPS uses larger numbers.

35. (***) Refer to Example 6.6. Write a Go program to compute and verify that indeed, $920^{17} \bmod 2773 = 948$ and $948^{157} \bmod 2773 = 920$.
36. Which of the following is NOT an example of personally identifiable information (PII), when discussing privacy protection?

    (a) The password of a student's personal computer
    (b) A student's full name
    (c) A student's university ID number
    (d) A student's full face photo

37. Zhang Lei is a product designer at a company which produced a product that is used by millions of users. Zhang discovers a bug in the product, which hackers could exploit to cause harm. What should she do according to the ACM Code of Conduct?

    (a) Do nothing, since she can work with colleagues to fix the bug in a couple of weeks.
    (b) Follow the practice of *responsible disclosure*.
    (c) Follow the practice of *full disclosure*. That is, announce the bug to the public without the consent of the company.
    (d) Report the bug to governmental regulators.

38. Refer to Example 6.9. Is the act of releasing the Morris worm wrong, according to the ACM Code of Conduct?

    (a) No, because Morris was just doing his research work, intending no harm.
    (b) No, because he was an outstanding academic, as demonstrated by his later achievements.
    (c) No, because the damage-causing bug was an accident. Nobody can guarantee that a sophisticated program is bug free.
    (d) Yes, because he was convicted and had served his sentence.
    (e) Yes, because his action did not do enough to avoid harm, contravening Principle 2 of the ACM Code of Conduct.

6.6 Bibliographic Notes

The chapter quotation on the original spirit of the World Wide Web is from an invited talk given by Tim Berners-Lee at the 2019 EmTech China Conference [1]. An accessible source of introductory Web programming is [2]. Hockney's formula on latency and bandwidth is discussed by Roger Hockney in [3]. A recent report on bandwidth growth trends of optical fibers can be found in [4]. Metcalfe's law, Reed's law, and evidence based on real data can be found in [5, 6]. References [7, 8] discussed the social network example of a rural video influencer. References [9, 10] discussed software-based and hardware-based cybersecurity examples. The RSA method example is based on material from [11, 12]. This book's description of professional norms referenced ACM Code of Ethics and Professional Conduct, the entry webpage of which is [13]. Three databases offer open access to genomic data on COVID-19 viruses [14–16].

References

1. Berners-Lee T (2019) Invited Talk at the EmTech China Conference hosted by MIT Technology Review, Beijing
2. https://www.w3schools.com/
3. Hockney RW (1996) The science of computer benchmarking. Society for Industrial and Applied Mathematics, Philadelphia
4. Hecht J (2016) Great leaps of light. IEEE Spectr 53(2):28–53
5. Metcalfe B (2013) Metcalfe's law after 40 years of Ethernet. Computer 46(12):26–31
6. Zhang XZ, Liu JJ, Xu ZW (2015) Tencent and Facebook Data validate Metcalfe's Law. J Comput Sci Technol 30(2):246–251
7. Jewell JR (2020) YouTube Commentary: "李子柒 Liziqi" — Nature and Internet Celebrity in the Time of the Coronavirus. The Arts Fuse. https://artsfuse.org/194752/youtube-commentary-李子柒-liziqi-nature-and-internet-celebrity-in-the-time-of-the-coronavirus/
8. Zhang X (2020) The rural video influencers in China: on the new edge of urbanization. Master of Arts Thesis, Cornell University
9. Cohen F (1987) Computer viruses: theory and experiments. Comput Secur 6(1):22–35
10. Lipp M, Schwarz M, Gruss D et al (2020) Meltdown: reading kernel memory from user space. Commun ACM 63(6):46–56
11. Rivest RL, Shamir A, Adleman L (1978) A method for obtaining digital signatures and public-key cryptosystems. Commun ACM 21(2):120–126
12. Zimmermann P (2020) Factorization of RSA-250. https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html
13. https://www.acm.org/code-of-ethics
14. https://www.ncbi.nlm.nih.gov/sars-cov-2/
15. https://www.gisaid.org/
16. https://bigd.big.ac.cn/ncov/?lang=en

Chapter 7
Projects

This chapter describes four practice projects, which offer students the opportunity to design an abstract computer, a real computer, and two computer applications. One of the applications creates a Web application. The overall design of the four projects is illustrated in the following Table 7.1. The project order can be adjusted. For instance, Project 3 can go before Project 2.

The projects also involve responsible computing practices, based on the principles of the ACM Code of Conduct. These projects emphasize independent work, collaboration, and acknowledgment. All used intellectual work should be acknowledged. A student can interact with others in doing the project, but the final product should be his/her own work.

More detailed projects information, based on the practice of the CS101 course at UCAS, can be found in Supplementary Material at the companion website.

The Supplementary Material also includes a sample learning/teaching platform, which includes the following resources:

- Software for students to do homework assignments (exercises) and have them checked and scored.
- Software for students to hand in project reports.
- Instructions and resources for students to download to their personal computers. Students are instructed to create a Linux environment for the course, including all necessary code and data files.
- A software tool to do the Turing Adder project.
- A software tool for the Text Hider project. It includes skeleton code, which serves as a starting point for the students to work on. The code follows good programming practices.
- A website for the Personal Artifact project, which includes a library of dozens of dynamic webpages from the teaching team and previous students.

Z. Xu, J. Zhang, *Computational Thinking: A Perspective on Computer Science*,
https://doi.org/10.1007/978-981-16-3848-0_7

Table 7.1 Overall design of the four projects

Project	Content	Purpose	Work mode
Project 1	Turing Adder	Design an abstract computer Appreciate Turing machine	Mostly work alone
Project 2	Text Hider	Design a computer application Appreciate programming	Mostly work alone
Project 3	Human Sorter	Design a real computer Appreciate how algorithm, software and hardware work together	Team work
Project 4	Personal Artifact	Appreciate creative expression Design a network application of dynamic webpage	Mostly work alone

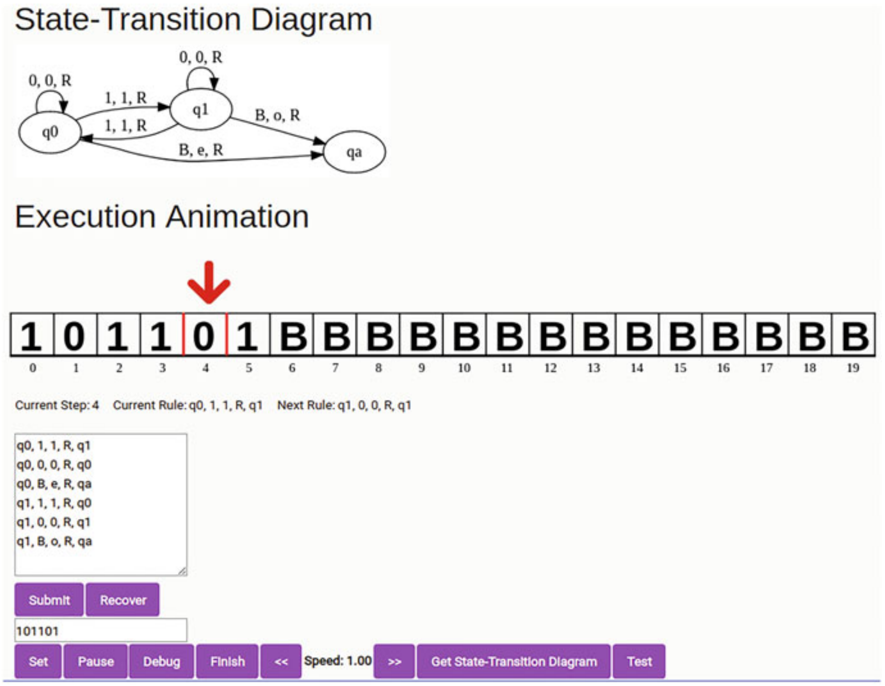
7.1 Turing Adder: Turing Machine for Serial Additions

The Turing Adder project augments students' understanding of an abstract computer: the Turing machine. It asks a student to do unary addition of 4-bit numbers, binary addition of 4-bit numbers, and binary addition of arbitrary-bit numbers. They are all bit-serial adders of unsigned positive numbers.

- **Objective**. Use the software tool provided by the teaching/learning platform, design a Turing machine by entering its state-transition table, for each of the following three computational tasks. A snapshot of the software tool used at UCAS is shown in Fig. 7.1.
  - Unary addition of two unary numbers, each of which is at most 4-bit long. For instance, 11 + 111 = 11111. That is, 2 + 3 = 5. Note that the result can be longer than 4 bits.

    Binary addition of two binary numbers, each of which is 4-bit long. For instance, 1011 + 0111 = 11010, or 11 + 7 = 18. Note that the result can be longer than 4 bits.
  - Binary addition of two binary numbers, each of which is of an arbitrary length between 4 and 64 bits. The platform tool automatically selects three lengths and checks the Turing machine for correctness.
- **Material and Method**. The main material to reference is Sect. 3.2.3 and the software tool from the teaching/learning platform.

  For each of the three tasks, we suggest students to adopt the following design and development procedure.
  - A student first develops the Turing machine on his/her personal computer, which is connected to the platform.
  - The student can see the corresponding state transition diagram of the Turing machine. Some students prefer such a graphic representation of a Turing machine, especially when the number of states is small.



Current Step: 4 Current Rule: q0, 1, 1, R, q1 Next Rule: q1, 0, 0, R, q1

```
q0, 1, 1, R, q1
q0, 0, 0, R, q0
q0, B, e, R, qa
q1, 1, 1, R, q0
q1, 0, 0, R, q1
q1, B, o, R, qa
```

Fig. 7.1 Snapshot of an example platform tool for the Turing Adder project

– The student can see execution animation of a Turing machine, as well as debug the design, either in a step-by-step mode or in a run-to-completion mode.
– After the student is sure about the correctness of the Turing machine, he/she can submit the work via the platform tool. Submission cannot be reversed.

• **Project Report**. Students do not need to turn in any project report. The project completes once all three Turing machines are submitted. The platform tool automatically checks for duplications.
• **Scoring**. Each student is scored for correctness of the three Turing machines.

– 50% for the 4-bit unary adder
– 20% for the 4-bit binary adder
– 30% for the arbitrary-bit binary adder

7.2 Text Hider: Program to Hide Text in Picture

The Text Hider project represents a computer application. It hides the content of the text file hamlet.txt in the picture of the image file Autumn.bmp. This is done by a Go program hide-0.go, which stores the modified picture in another image file doctoredAutumn.bmp. Students also need to develop a Go program show-0.go, to recover the content of the text file hamlet.txt from doctoredAutumn.bmp.

- **Objective**. Develop a Go program to hide the content of a text file in an image file, as well as a Go program to recover the content of the text file from the doctored image file. The basic principle of hiding is to replace the least significant two bits of one byte of the Pixel Array by two bits of a character, as shown in Example 5.12. The objective has four detailed interpretations:
  - The doctored image file must show no visible difference from the original image file, as demonstrated in Example 5.12.
  - The two programs should work for other text and image files, assuming the BMP image file format. In other words, it should work if we want to hide the content of the text file aMiddleSummersDream.text in the picture of the image file Spring.bmp.
  - This project emphasizes good programming practice, as illustrated in UKA Unit 6 in Sect. 2.2. Both hide-0.go and show-0.go should follow such practices.
  - This project emphasizes independent work. Each student should complete this project on his/her own. To help check for work independence, students must use the two files, hamlet.txt and Autumn.bmp, which are provided by the platform. Use by other means, such as borrowing a file from a fellow student through a USB stick, could cause the project to fail, even without the student being alerted.

Material and Method. Most material is already provided in Example 5.12. Recall that the algorithm for hide-0.go is as follows:

Algorithm for hide-0.go

- **Input**: A text file hamlet.txt and an image file Autumn.bmp.
- **Output**: A doctored image file doctoredAutumn.bmp
- **Steps**:

```
1. Read Autumn.bmp into variable p        // p for picture
2. Read hamlet.txt into variable t         // t for text
3. Hide the length of hamlet.txt in the first 32 bytes of Pixel Array
4. Hide hamlet.txt in variable p in the remaining bytes of Pixel Array
5. Write p to file doctoredAutumn.bmp
```

Students need to pay attention to the following items:

- Design and develop the hide-0.go program to follow good programming practices. The code in the textbook is incomplete and does not pay much attention to good programming practices.
- Design an algorithm for show-0.go.
- Develop the show-0.go program and follow good programming practices.
- Verify that the two programs indeed hide and recover a text file in an image file, by executing them on the baseline files hamlet.txt and Autumn.bmp.

- **Project Report**. Every student needs to turn in a project report, including:
  - Description of the student's own design on how to hide and show
  - Source code of hide-0.go and show-0.go
  - Evaluation of programs' executions
  - Reflection and discussion, including any unusual happenings
  - Acknowledgements, if any
- **Scoring**. Each student is scored for programming (including good programming practices) and independent work.
  - 85% for the student's design and code, if achieving the project objective
  - 15% for communications clarity of the project report

7.3 Human Sorter: Team Computer for Quicksort

The Human Sorter project invites students to design a real computer: a computer made of a team of students to do quicksort. By designing and testing this team computer, students learn team work. Each student also reports to the team how computer hardware, instructions, and program execution work as a whole.

- **Objective**. Design a team computer of students to execute a quicksort program, to sort the students in the team from order by name to order by height (Fig. 7.2).

  Each student needs to produce a design, including:
  - The team computer organization
  - The instruction set of the team computer
  - The quicksort program made of a sequence of such instructions
  - The evaluation record of program executions

    The evaluation must show that the design satisfies three correctness properties:
  - *Result correctness*: the students are indeed ordered by height.
  - *Algorithmic correctness*: the execution implements the quicksort algorithm.
  - *Systems correctness*: the team computer executes the program sequentially, i.e., step-by-step, one instruction after another.

- **Material and Method**. A CS101 course usually enrolls hundreds of students. This large class is divided into small teams. Each team needs a team leader and should consist of no more than 30 students. Remember that the quicksort algorithm takes at least O(n log n) steps.
  - Data, memory, processor are all made of humans (students). The team computer is a human computer, not an electronic computer.
  - The team computer hardware organization must consist of memory and processor. No I/O devices are necessary.

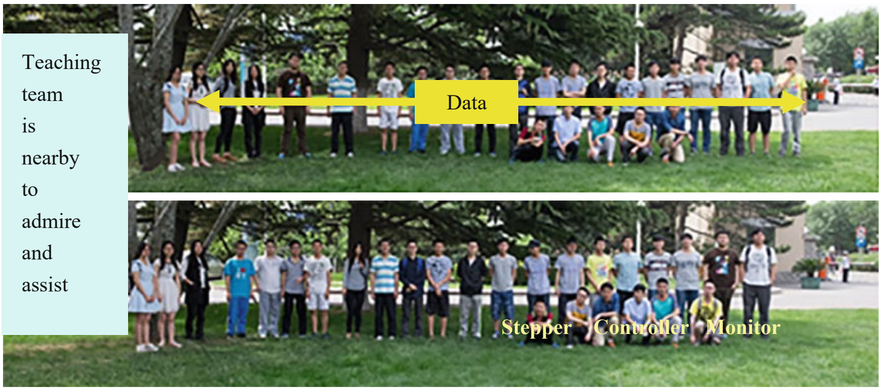


Fig. 7.2 A team computer sorts a team of students: from order by name to order by height. (Photos are blurred for privacy. Photos credits: Haoming Qiu)

- For tips on computer hardware design, refer to Sects. 2.3 and 5.3.3. But, forget not that this is a human computer. Each student and each team can use their imagination in the design. It does not have to be a computer following the von Neumann architecture. For instance, the instructions of the quicksort code do not have to be stored in the memory of the team computer.
- For tips on instruction set design and the quicksort program implementation, refer to Sects. 2.3, 4.3.3 and 5.3.3.
- It is best that program executions are conducted on a sufficiently large open space, such as on a big lawn that can accommodate hundreds of students.
- Each team must keep meticulous records of execution, down to the execution of every instruction step. For each execution of the quicksort program, the initial configuration, the final configuration and the number of steps are reported. That is one reason why the team computer in Fig. 7.2 has a *monitor*, who keeps the record of every execution step. The *stepper* serves as the system clock, to count the steps and to make sure that the rest of the system follows the step-by-step rule. *Humans tend to violate this rule*.

• **Project Report**. Every student needs to report his/her design to the team. Each team comes up with its design by selecting and combining designs from the team members, through a process of *rough consensus and running code* (the IETF mantra). At the end of the project after the team's execution runs, each student hands in a project report, which contains the following contents:

  - The student's own design, which could be a revised version built on the team's design
  - Evaluation of the team's design, with sufficient evidence
  - Reflection and discussion, including any unusual or funny happenings
  - Acknowledgements

Note that there is no project report handed in by a team. However, each team should present its design to the entire class of hundreds of students, by class presentation, posters, or webpages.

In the UCAS CS101 course, this project normally takes 3–4 sessions:

– Session 1. Individual students go through their designs
– Session 2. Teams fix their designs
– Session 3. Teams run their quicksort programs on their team computers
– Session 4. Teams report their projects to the entire class

• **Scoring**. Each student is scored for design, team work, and communication.

  – 80% for the student's design, if it achieves the project objective
  – 10% for the team leader's report
  – 10% for communications clarity of the project report

7.4 Personal Artifact: Web Page of Creative Expression

This Personal Artifact project encourage students to demonstrate their creative expression by designing a dynamic webpage. Students are also encouraged to learn any additional knowledge needed by themselves.

• **Objective**. Design a dynamic webpage of creative expression. Successful completion of this personal artifact needs students to demonstrate their self-learning capability, as the textbook or the lecturers do not cover much Web programming material, except that in Sect. 6.3.2.

  Each student needs to produce and show to the class a webpage:

  – A dynamic webpage including HTML, CSS, and JavaScript code
  – A webpage showing creative expression

• **Material and Method**. The students each use the material in Sect. 6.3.2 as a starting point, create a dynamic webpage, and upload the files and the project report to the class website. In the process, students may need to reference the following sources of resources.

  – The Internet. For instance, additional Web programming knowledge can be found at https://www.w3schools.com/.
  – The library of Personal Artifacts examples in the Supplementary Material. Previous students have created a dozen genres. The top three are scientific artifacts, games, and artistic communications.

  Three points should be emphasized in producing the artifact.

– Most of the artifact should be made by the student.
– Used material from other sources should be properly acknowledged.

- Make an effort to control the program size of your webpage. 100–200 lines of HTML/CSS/JavaScript code can go a long way. Avoid copying existing code into your product.

Some students created webpages for “My Beautiful Homeland”. The idea is wonderful. Some students created a webpage with links to beautiful photos enticing people to visit their homeland as tourists. However, such projects could get a low score for two problems: (1) the webpage is not dynamic, and (2) most photos are creations of other people.

- **Project Report**. The project report has a free form, including:
  - Articulation of the artifact, including reflection and discussion
  - Source code of the webpage program
  - Acknowledgements
- **Scoring**. Each student is scored for design and communication.
  - 90% for the student’s design, if it achieves the project objective
  - 10% for communications clarity of the project report

Chapter 8
Appendices

8.1 Multiples and Fractions

Bse 10	Base 2	Symbol	Prefix	Example
10E24	2E80	Y	Yotta	
10E21	2E70	Z	Zetta	ZB, zettabytes, the world's data volume
10E18	2E60	E	Exa	EFLOPS, Exa floating-point operations per second, speed of supercomputers
10E15	2E50	P	Peta	PB, petabytes, a server's storage capacity
10E12	2E40	T	Tera	TB, terabytes, a disk's storage capacity
10E9	2E30	G	Giga	Gbps, giga bits per second, bandwidth of a local area network
10E6	2E20	M	Mega	MW, Mega Watt, power consumption of a supercomputer
10E3	2E10	K	Kilo	Kg, Kilogram, weight of a laptop computer
10E2		H	Hecta	
10E1		da	deca	
10E-1		d	deci	100 ms, 100 milliseconds, good interaction time when using a computer
10E-2		c	centi	
10E-3		m	milli	mm, millimeter, size of a semiconductor die
10E-6		μ	micro	μs, microsecond, communication latency between two nodes of a supercomputer
10E-9		n	nano	nm, nanometer, feature size of a semiconductor technology
10E-12		p	pico	pJ, picojoule, energy consumption of an arithmetic operation in a computer
10E-15		f	femto	fs, femtosecond, laser wavelength
10E-18		a	atto	
10E-21		z	zepto	Landauer's principle: Erasing one bit needs zeptoJoule energy

Note that the same symbol may imply different values using base 10 or base 2.

Z. Xu, J. Zhang, *Computational Thinking: A Perspective on Computer Science*,
https://doi.org/10.1007/978-981-16-3848-0_8

T = 1.00 × 2E40 = 1,099,511,627,776	≈1.10 × 10E12	≠1.00 × 10E12
G = 1.00 × 2E30 = 1,073,741,824	≈1.07 × 10E9	≠1.00 × 10E9
M = 1.00 × 2E20 = 1,048,576	≈1.05 × 10E6	≠1.00 × 10E6
K = 1.00 × 2E10 = 1,024	≈1.02 × 10E3	≠1.00 × 10E3

This difference is the reason why some users thought that a vendor may give them fewer resources in a computer product. For instance, a 1-TB hard disk may actually have only a storage capacity of 10E12=1,000,000,000,000 bytes, not 2E40=1,099,511,627,776 bytes, a shortage of about 100GB.

8.2 Programming Basics

Programming in the Go programming language (Golang) is explained in preceding chapters. For ease of reference, we summarize the programming constructs in four tables: shell commands, packages, statements, and data types (Tables 8.1, 8.2, 8.3, and 8.4).

In many Go programs of this book, e.g., fib.dp.bu.go in Example 34 of Section 4.3.1, input data are hardwired into the code, to simplify the code. This is bad programming practice. The fib.dp.bu.go code only works for F(5). It is better to change

Table 8.1 Commands in a Linux shell used in this book

Command	Purpose	Example
cat	Print file to standard output	>cat hello.go print hello.go to screen
cd	Change directory	>cd .. change to the parent directory
display	Display a picture	>display ucas.bmp display ucas.bmp on screen
./hello	Execute binary code hello in current directory	
go build	Compile Go source file into executable file	>go build hello.go compile hello.go into executable file hello
go run	Compile and execute Go program	>go run hello.go compile and execute hello.go
ls	List files in current directory	>ls . list files of current directory
Tab key	Automatically complete a command	
↑	Execute the previous command	
Ctr-C	Exit the current command	
Ctr-S	Save the file in editing	

Table 8.2 Golang packages used in this book

Package	Example

fmt	For input and output functions, e.g.,
	fmt.Println("Hello") //print "Hello" to display
	fmt.Printf("One=%d",1) //print "One=1" to display
	fmt.Scanf("%d",&A) //enable user to enter integer to variable A

io/ioutil	For reading/writing files, e.g.,
	p, _ := ioutil.ReadFile("./ucas.bmp") //read data in ucas.bmp to variable p
	ioutil.WriteFile("./mucas.bmp", p, 0666) //write p to mucas.bmp

math	For mathematical functions, e.g., math.Pow(2,3) //returns 2^3=8
os	For interaction with operating system, e.g., V := os.Args[0] //os.Args is an array of command parameters

```
  func main() {
    fmt.Println("F(5)=", fibonacci(5))
  }
```

in fib.dp.bu.go to the following code, which enables user to enter a number.

```
 func main() {
  var n int = 0
  fmt.Printf("Please enter a natural number between 0 and 92: ")
  _,err := fmt.Scanf("%d", &n)        // n = user-entered integer
  if err != nil {
   fmt.Println("Input Error: Not a number")
   return
  }else if n < 0 {
   fmt.Println("Input Error: Please enter a non-negative integer.")
   return
  }else if n >92 {
   fmt.Println("Input Error: The number is too large. The program
overflows.")
   return
  }
  fmt.Printf("F(%d) = %d\n", n, fibonacci(n))
 }
```

8.3 Pointers to Supplementary Material

The companion website cs101.ucas.edu.cn provides supplementary material for (1) lecture and projects slides, (2) the source code of programs, and (3) answers to even-numbered homework exercises. In addition, it contains pointers to sample programs and tools which help provide a teaching and learning platform. Among other code, the following are included.

Table 8.3 Golang statements used in this book

Statement	Example
Assignment	v := 1 // assign 1 to v a,b := true,false // assign true to a and false to b

Break	Terminate a loop, e.g.,
	for i:=0;i<5;i++{
	if(i>=3) break
	fmt.Printf("%d ",i)
	}
	The code above will print 0 1 2, because the loop is terminated by break when i==3

Continue	Go to beginning of the next loop iteration, e.g.,
	for i:=0;i<5;i++{
	if(i<3) continue
	fmt.Println("%d ",i)
	}
	The code above will print 3 4, because when i<3, fmt.Println() is skipped

Declaration	v := 3 // declare a variable v and assign 3 to it var v int = 3 const c = 3 // declare a constant c and assign 3 to it const c int = 3

For loop	// Compute sum of an integer array
	sum := 0
	var arr [5]int = [5]int{0,1,2,3,4}
	for i:=0; i<len(arr); i++{ // i:=0; is init statement, i<len(arr); is
	sum += arr[i] // condition and i++ is post statement
	}

Function definition	// Define an addition function
	func Add(a int, b int) int { // Add is the function name
	return a+b // a,b are parameters of the function
	}

Function call	c := Add(1,2) // call Add function to obtain c = 3

If statement	if a<b {
	fmt.Println("Smaller") // if a<b, this statement will be executed
	}else{
	fmt.Println("Not smaller") // if a>=b, this statement will be executed
	}

Return	Specify the return value of a function, e.g., "return a+b" in Add

- Go programs:

```
binary.search.go
fib-10.go
fib-5.go
fib-50.go
fib.binet-50.go
fib.dp-5.go
fib.dp-50.go
fib.dp.big.go
fib.dp.go
```

Table 8.4 Go data types used in this book

Data type	Example
array	var a [10]int // define an array consisting of 10 integers var primes [3]int = [3]int {2,3,5} // define an array consisting of 2,3,5 var p0 int = primes[0] // primes[0] is the zero-th element of primes
bool	var a bool = true // define a bool variable named "a", whose value is true b,c := true,false // define 2 bool variables whose values are true,false
byte	var X byte = 'a' // define byte variable X, assign ASCII encoding of 'a' to it
int	var y int = 1 // define a signed integer variable whose value is 1

slice	var prime_array [3]int = [6]int {2,3,5} // prime_array is an array
	var s []int = primes[0:2] // s is a slice representing the first 2 elements of primes
	var u []int = make([]int,3) // u is a slice representing a nameless array consisting of 3 integers

string	var str1 string = "directly declaration" // define a string var n int = len(str1) // len() returns the number of characters in str1
uint	var i uint = 1 // define an unsigned integer variable whose value is 1

```
fib.dp.bu.go
fib.go
fib.matrix.go
fib.Uint.go
hash.search.go
hello-1.go
hello.go
hide-0.go
linear.search.go
name_to_number-0.go
name_to_number-1.go
name_to_number.go
null.go
parity.go
pi.go
pointer.go
replace.go
symbols.go
testPoint123.go
WebServer.go
```

- Web programs:

```
myFirstWebPage.html
staticChildrensDay.html
ChildrensDay.html
```

Index