# Computational Thinking:
# A Perspective on Computer Science

Zhiwei Xu  Jialin Zhang
University of Chinese Academy of Sciences

2021.03.06

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 is based on 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, including lecture slides and project software, is provided at cs101.ucas.edu.cn.

Zhiwei Xu
Jialin Zhang
Beijing, China
January 2021

# 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. 0.1. 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 four years designing the course and six years teaching the material. The contents have gone through three major revisions. For instance, version 1 had no programming. Version 2 included Go language programming contents. Version 3 (the current version) requires students to write roughly 300 lines of Go code and 100 lines of Web code, where most students can learn Web programming by themselves.

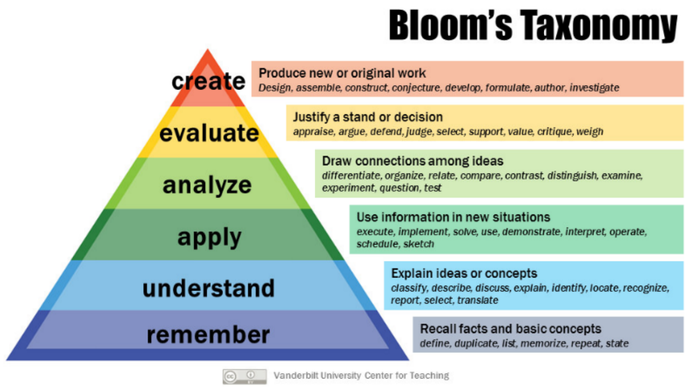
Bloom's Taxonomy
create
evaluate
analyze
apply
understand
remember
Produce new or original work
Design, assemble, construct, conjecture, develop, formulate, author, investigate
Justify a stand or decision
appraise, argue, defend, judge, select, support, value, critique, weigh
Draw connections among ideas
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. 0.1** 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. 0.2.

- 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. 0.2a. 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. 0.2b. 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 1 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. 0.2c. 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 three days. Half of her time was spent on thinking, designing, and making. Coding and debugging accounted for no more than 50% of the time.

(a) Sort a class of students: from an order by name to an order by height

(b) What's the area size of the panda? (c) Part of the *Kitty Band*

**Fig. 0.2** Three examples of problem-solving and creative expression
Photos and graphics credits: Haoming Qiu, Hongrui Guo, Siyue Li

## 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 0.1.

**Table 0.1** A sample course schedule for the Spring semester of the year 2020 at UCAS

| Week | Lecture<br>Two classes per week | Project<br>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 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, compleXity, **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 the 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.

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 six-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. 0.1, 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 six 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 get 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 students are 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 0.2 and 0.3, 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 CS 101 as an on-line course for the Spring semester of 2020 (Table 0.1). 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 on-line courses could be more efficient.

**Table 0.2** Suggested schedule for a 3-credit course

| Week | Lecture<br>Two classes per week | Project<br>Two classes per week | Due Date<br>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 0.3** Suggested schedule for a 2-credit course

| Week | Lecture<br>Two classes per week | Project<br>Two classes per week | Due Date<br>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, 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 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.** $(110.101)_2 = (?)_{10}$

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

☶

## Supplementary Material

The companion website **cs101.ict.ac.cn** provides supplementary material for (1) lecture and projects slides, (2) text, graphics, sound, and video files, and (3) sample programs for teaching and learning. Besides, the Appendices of the textbook contains pointers to the source code of all programs and solutions to even-numbered homework exercises.

## Acknowledgments

We are grateful to many people for feedbacks 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 Pekin University has provided persistent encouragements and feedbacks. 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 ten 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), Xiaomi. ACM and IEEE-CS are international societies of computer professionals. CCF is the society of computer professionals in China with over 60 thousand 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 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].

[1] D'Agostino S. The Computer Scientist Who Can't Stop Telling Stories. Quanta Magazine. April 16, 2020. https://www.quantamagazine.org/computer-scientist-donald-knuth-cant-stop-telling-stories-20200416.

[2] Bloom B S, Engelhart M D, Furst E J, et al. Taxonomy of educational objectives: Cognitive domain. Longman Group, 1956.

[3] Anderson LW, Krathwohl DR, Airasian PW, Cruikshank KA, Mayer RE, Pintrich PR, Raths J, Wittrock MC. A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives, abridged edition. White Plains, NY: Longman. 2001.

[4] Wing J M. Computational thinking. Communications of the ACM, 2006, 49(3):33-35.

[5] US National Research Council. Computer Science: Reflections on the Field, Reflections from the Field[M]. Washington D.C: National Academies Press, 2004.

[6] College Board, AP Computer Science Principles Course and Exam Description. https://apcentral.collegeboard.org/pdf/ap-computer-science-principles-course-and-exam-description.pdf, 2020.

[7] Page D, Smart N. What Is Computer Science?: An Information Security Perspective. Switzerland: Springer International Publishing, 2014.

[8] Schneider G M, Gersting J. Invitation to Computer Science, 8th Edition. Cengage Learning, 2018.

[9] Dale N, Lewis J. Computer Science Illuminated, 7th Ed. Jones & Bartlett Learning, 2019.

[10] Brookshear G, Brylow D. Computer Science: An Overview, 13th Ed. Pearson, 2019.

[11] Alvarado C, Dodds Z, Kuenning G, Libeskind-Hadas R. CS for All: An Introduction to Computer Science Using Python. Franklin, Beedle & Associates Inc. 2019.

# 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 Chapter 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, 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.

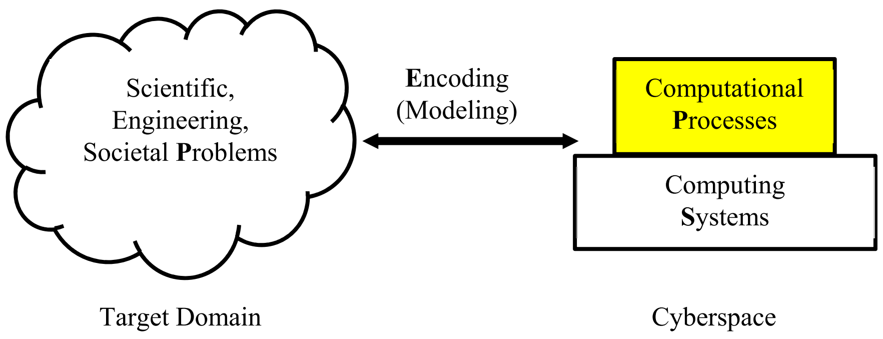

**Fig. 1.1** Computational processes in problem solving: the PEPS model

- **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 Section 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.
- 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. 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.

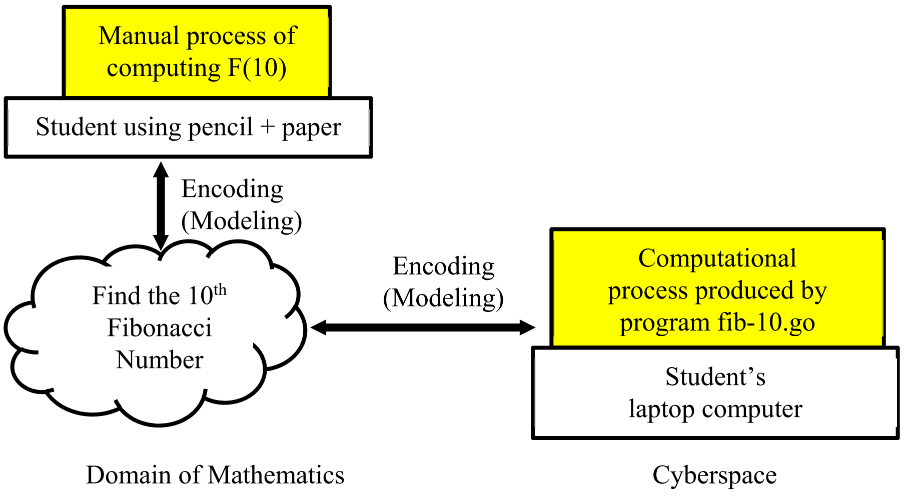
Manual process of computing F(10)
Student using pencil + paper
Encoding (Modeling)
Find the 10th Fibonacci Number
Encoding (Modeling)
Computational process produced by program fib-10.go
Student's laptop computer
Domain of Mathematics
Cyberspace

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

**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,
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.

```
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) An algorithm to find F(10) directly from the mathematical definition

```
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) A Go program fib-10.go that implements the algorithm

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

(c) Compile fib-10.go and execute fib-10 to produce the output

**Fig. 1.3** Computational process for finding the 10th Fibonacci number F(10)
Texts after a double slash (//) are **comments** to explain the code

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 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 various syntactic errors, called **compile-time 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 the command, not the command itself. An operating system such as Linux normally provides a command interpreter called **shell**.

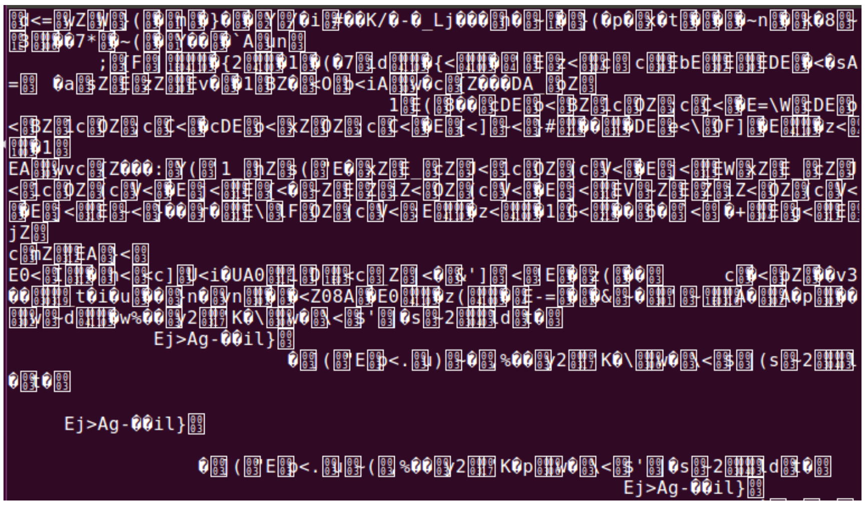
**Fig. 1.4** Screen display of the machine code fib-10: scrambled symbols

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 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 ore more functions. For instance, the size of the area in Fig. 4a 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. 4b 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. 1.4c, 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.

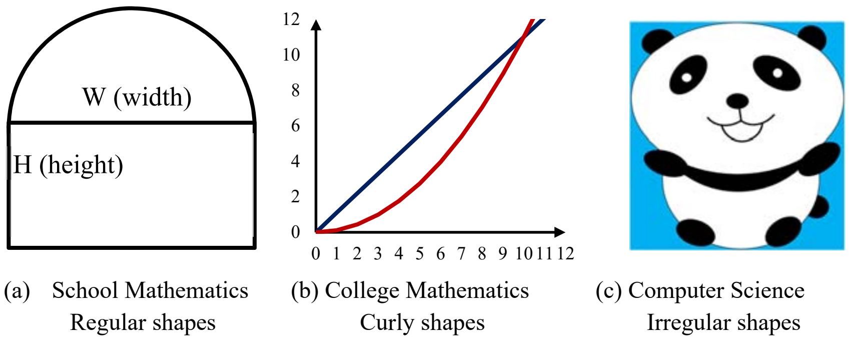

**Fig. 1.5** Computer science enables us to compute sizes of irregular shapes

*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 nine 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.

**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. |

**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 2. Computing larger Fibonacci numbers F(50) and F(100)**

The manual calculation process of Fibonacci numbers in Example 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 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 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 | |
| | = 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** | |

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.

```
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) An algorithm to find F(50) directly from Binet's formula

```
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) A Go program fib.binet-50.go that implements the algorithm

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

(c) Compile fib.binet-50.go and execute fib.binet-50 to produce the output

**Fig. 1.6** Using Binet's formula to compute the 50th Fibonacci number F(50)

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)))/sqrt5

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 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 Chapter 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 5 below, this program fib.dp-50.go is smarter and much faster than the program fib-50.go in Example 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 5. Contrasting four processes to find Fibonacci number F(50)

The manual calculation process for Fibonacci numbers in Example 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 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 Chapter 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, and the three automatic processes using fib-50.go in Example 2, fib.binet-50.go in Example 3, and fib.dp-50.go in Fig. 2.9, respectively.

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 minutes 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.

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

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

## 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. 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 Chapters 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. 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 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 Chapter 5.

We use an example below to illustrate the objectives of logic thinking, algorithmic thinking, and systems thinking. The details will be discussed in Chapters 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 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 3, we see that running fib.dp-500.go generates a wrong output.

```
F(500)= 2171430676560690477          by fib.dp-500.go
F(500)= 1.3942322456169767e+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.

**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 |

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 seconds. 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 seconds, or a little more than three hours, 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 summarized the fundamentals of computer science, including the essential character and salient characteristics, from researchers' viewpoint. These fundamental concepts are listed in Box 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 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:

- 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*.

**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 |

## 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 everyday. 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.

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.

**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% |

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 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 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 4. We have Nordhaus's law for computer 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 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 two 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).

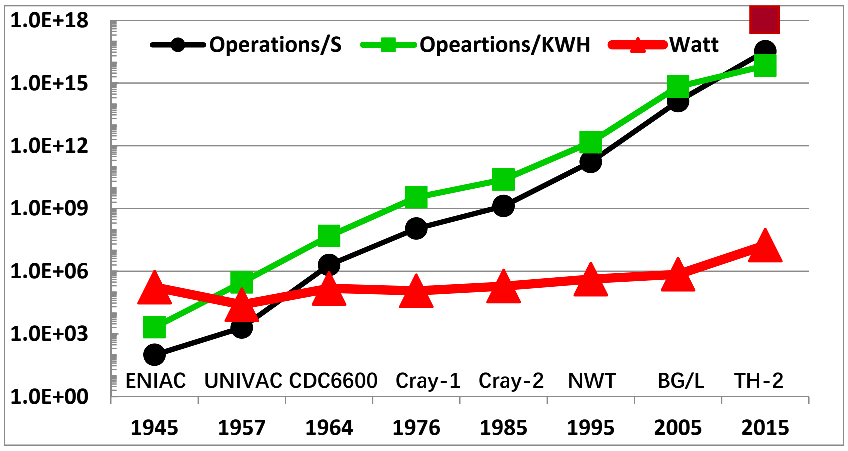

**Fig. 1.7** Growth trends of computing speed, energy efficiency, and power consumption of the world's fastest computers (supercomputers) from 1945 to 2015. Special thanks to Drs. Gordon Bell, Jonathan Koomey, Dag Spicer and Ed Thelen for providing data for the first three computers.

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 $n = 300\sim600$). 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 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. 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 two 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 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, 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 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.

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

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 seventy-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 6 shows part of the verse named *Cha* (差, ternary symbol 𝌐), translated roughly as *diversity* or divergence.

**Box 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 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(50)= 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. Refer to Example 1 and Fig. 1.3.
   (a) Designing the algorithm in Fig. 1.3 is not a computational process, since the design process is done by human, not automatically executed by a computer.

(b) Designing the algorithm in Fig. 1.3 is not a computational process, since the design process is not a step-by-step process of information transformation.
(c) The process of programming-compilation-execution is a computational process, since it is a step-by-step process of information transformation. The programming step transforms the algorithm into the high-level language program fib-10.go. The compilation step transforms the high-level language program fib-10.go into the machine code fib-10. The execution step transforms the machine code fib-10 into the output F(50)= 55.
(d) The process of programming-compilation-execution is not a computational process, since it is not fully automatic. Programming, entering the compilation command, and entering the execution command are done by human.

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

(e) F(10)= 55
(f) F(10)= 12586269025
(g) F(50)= 55
(h) F(50)= 12586269025

9. Refer to Example 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. 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 NOT 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.

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) 4 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.

[1] Susan D'Agostino. The Computer Scientist Who Can't Stop Telling Stories. Quanta Magazine. April 16, 2020. 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. Personality and Social Psychology Bulletin, 21(6), 558-571.

[3] Wing J M. Computational thinking. Communications of the ACM, 2006, 49(3):33-35.

[4] US National Research Council. Computer Science: Reflections on the Field, Reflections from the Field[M]. Washington D.C: National Academies Press, 2004.

[5] Huawei and Oxford Economics (2017). Digital spillover: Measuring the true impact of the digital economy. Available at: https://www.huawei.com/minisite/gci/en/digital-spillover/index.html.

[6] China Info 100 (2018). The 2017 China Digital Economy Development Report. Available at: http://www.chinainfo100english.com/201803/432.html.

[7] World Summit on the Information Society. Building the information society: a global challenge in the new Millennium[J]. Declaration of Principles, 2003.

[8] https://www.wikizero.com/en/Digital_infinity.

[9] Karp R M. Understanding science through the computational lens. Journal of Computer Science and Technology, 2011, 26(4):569-577.

[10] Yann Moulier-Boutang. Cognitive Capitalism and Entrepreneurship: Decline in industrial entrepreneurship and the rising of collective intelligence. Conference on Capitalism and Entrepreneurship. Cornell University, Ithaca, New York, September 28-29, 2007.

[11] Nordhaus W D. Two Centuries of Productivity Growth in Computing. Journal of Economic History, 2007, 67(1):128-159.

[12] Moore G E. Progress in digital integrated electronics. Electron devices meeting. 1975, 21:11-13.

[13] Hecht J. Great leaps of light. IEEE Spectrum, 2016, 53(2):28-53.

[14] Xu Z W, Chi X B, Xiao N. High-Performance Computing Environment: A Review of Twenty Years Experiments in China. National Science Review, 2016, 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. Communications of the ACM, 59(11), 105-112.

[16] Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Tunyasuvunakool, K., ... & Hassabis, D. (2020). High accuracy protein structure prediction using deep learning. Fourteenth Critical Assessment of Techniques for Protein Structure Prediction (Abstract Book), 22, 24.

[17] Strogatz S. The Real Scientific Hero of 1953. New York Times, March 4, 2003.

[18] Richards D F, Krauss L D, Cabot W H, 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 and www.youtube.com/watch?v=Wr7WbKODM2Q.
[19] Xu Z W, Li G J. Computing for the Masses. Communications of ACM, 2011, 54(10):129-137.
[20] Akiyama K, Alberdi A, Alef W, et al. First M87 event horizon telescope results. IV. Imaging the central supermassive black hole. The Astrophysical Journal Letters, 2019, 875(1): L4.
[21] Gray J. What next?: A dozen information-technology research goals. Journal of ACM, 2003, 50(1):41-57.
[22] Bell, G. (2008). Bell's law for the birth and death of computer classes. Communications of the ACM, 51(1), 86-94.
[23] Bush V. As we may think. The Atlantic Monthly, 1945, 176(1): 101-108.
[24] Bush V. Memex revisited. In Nyce J and Kahn P. From Memex to Typertext: Vannevar Bush and the Mind's Machine. Academic Press Professional, Inc., 1991: 197-216.
[25] Turing A M. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 1936–1937, 42(2):230-265.
[26] Turing A M. Computing Machinery and Intelligence. Mind, 1950, 59(236):433–460.
[27] Nylan M. The Canon of Supreme Mystery by Yang Hsiung: A Translation with Commentary of the T'ai Hsuan Ching. SUNY Press, 1993.

# 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, Figure 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 7th month (July), which will be elaborated in an exercise.

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

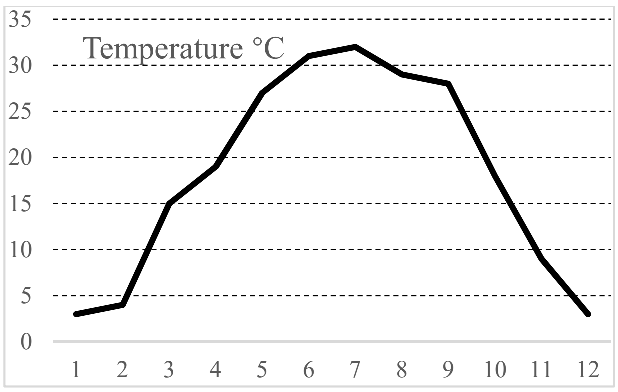

**Fig. 2.1** An analog quantity: average high temperature value in Beijing in year 2019

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.

## 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.

**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 |

**Example 8.** **$(110.101)_2 = (?)_{10}$**

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

☶

**Example 9.** **$(6.625)_{10} = (?)_2$**

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.

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** | | |

**Example 10.** **$(11.3)_{10} = (?)_2$**

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

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

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.

**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 |

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. 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 = **1**00000000 = $00000000_2 = 0_{10}$

$\underline{11000001}$

The carry bits **1**1111111

The result bits **1**00000000 = $00000000_2 = 0_{10}$

## 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).

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].

| $D_7D_6D_5D_4$ / $D_3D_2D_1D_0$ | 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

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.

### 4. 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.

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

(a) The simplest Go program null.go which is correct but does nothing

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

(b) A correct program hello.go which outputs hello!

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

(c) A wrong program hello-1.go which produces compiling error

**Fig. 2.3** Some simple programs

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
> 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 is 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                    ; Shell sends error
.\hello-1.go:2:6: missing function body     messages to StdErr
.\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.

## 5. 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.

```
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) Program symbols.go

```
> 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) Output by executing program symbols.go

**Fig. 2.4** Program symbols.go and its output

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.

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) Source code of program name_to_number-0.go

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

(b) Screen output by executing program name_to_number-0.go

**Fig. 2.5** Program name_to_number-0.go and its output

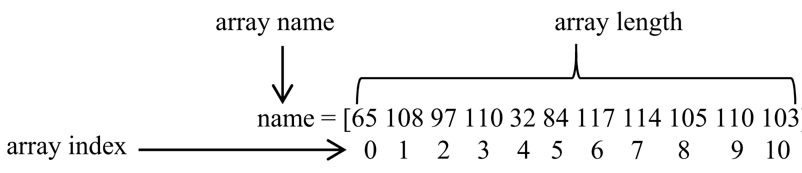

**Fig. 2.6** Illustration of an array variable, called name, after the declaration statement var name string = "Alan Turing"

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:

```
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.

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.

**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 |

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 11. 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 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.

```
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

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.

## 6. 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 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.

```
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

## 7. 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.

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.

```
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) Recursive fib-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
  }
  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) Dynamic programming fib.dp-50.go

**Fig. 2.9** The recursive and dynamic programming programs to compute Fibonacci numbers

Program fib-50.go is almost the same as fib-10.go in Example 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.

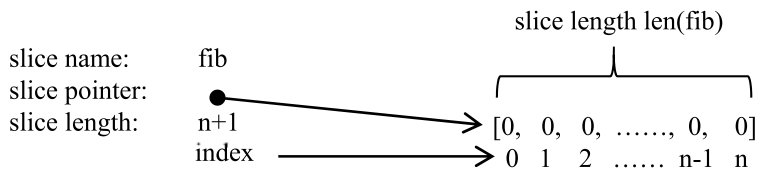

**Fig. 2.10** Illustration of a slice variable fib, after statement var fib []int = make([]int, n+1)

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.

- **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 and machine learning processing.
  - ■ 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.

[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.

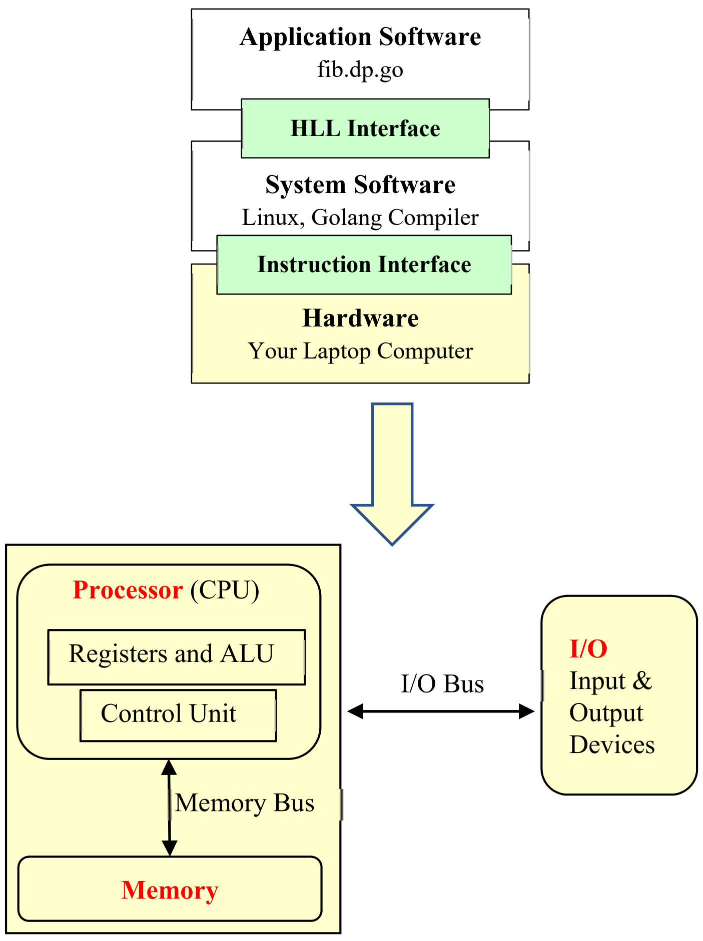

**Fig. 2.11** The stored-program model of computers, also known as the von Neumann model

## 8. 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.

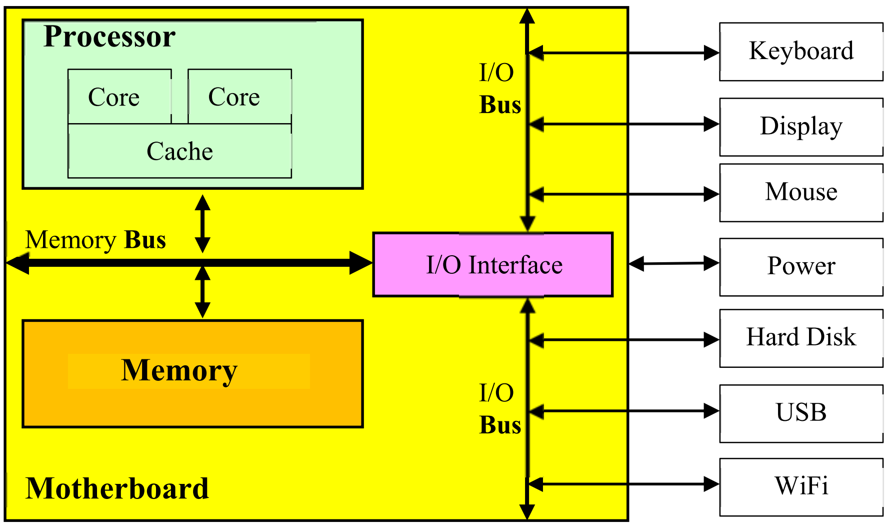

**Fig. 2.12** Illustration of how main components are organized to form a computer

**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, 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 **multi-core** 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?

**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 |

## 9. Putting it together: 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.

**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] |

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

```
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** 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 | |
| | | 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 Section 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 0°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, with a precision of one digit after the decimal point. In other words, we need to be able to display all temperature readings: -49.9, …, -00.1, 00.0, 00.1, …, 49.9. 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: 10 bits |
| (b) The simple signed integer format needs () | Y: 11 bits |
| (c) The two's complement format needs () | Z: 12 bits |

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 Section 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:

   (g) fmt.Printf("%c", '?')
   (h) fmt.Printf("%b", 63)
   (i) fmt.Printf("%c", 63)
   (j) fmt.Printf("%d", 63)
   (k) fmt.Printf("%c", ?)
   (l) 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 mod 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, Chapter 2 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?

(e) The address of fib[i-1] is R0+R2*8-8.
(f) The address of fib[i-2] is R0+R2*8-8.
(g) The address of fib[i-1] is R0+R2*8-16.
(h) 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.

[1] Simon, H. A., & Newell, A. (1976). Computer science as empirical inquiry: symbols and search. Communications of the ACM, 19(3), 11-126.

[2] Moye, W. T. (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 (pp. 257-259). Springer, Cham.

[4] https://tour.golang.org/welcome/1