1 Computing

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

Edsger Dijkstra, 1972 Turing Award Lecture

The first million years of hominid tool development focused on developing tools to amplify, and later mechanize, our physical abilities to enable us to move faster, reach higher, and hit harder. We have developed tools that amplify physical force by the trillions and increase the speeds at which we can travel by the thousands.

Tools that amplify intellectual abilities are much rarer. While some animals have developed tools to amplify their physical abilities, only humans have developed tools to substantially amplify our intellectual abilities and it is those advances that have enabled humans to dominate the planet. The first key intellect amplifier was language. Language provided the ability to transmit our thoughts to others, as well as to use our own minds more effectively. The next key intellect amplifier was writing, which enabled the storage and transmission of thoughts over time and distance.

Computing is the ultimate mental amplifier—computers can mechanize any intellectual activity we can imagine. Automatic computing radically changes how humans solve problems, and even the kinds of problems we can imagine solving. Computing has changed the world more than any other invention of the past hundred years, and has come to pervade nearly all human endeavors. Yet, we are just at the beginning of the computing revolution; today's computing offers just a glimpse of the potential impact of computing.

There are two reasons why everyone should study computing:

1. Nearly all of the most exciting and important technologies of today and tomorrow are driven by computing.
2. Understanding computing illuminates deep insights and questions into the nature of our minds, our culture, and our universe.

Anyone who has submitted a query to Google, watched *Toy Story*, had LASIK eye surgery, made a cell phone call, seen a Cirque Du Soleil show, shopped with a credit card, or microwaved a pizza should be convinced of the first reason. None of these would be possible without the tremendous advances

in computing over the past half century.

It may be true that you have to be able to read in order to fill out forms at the DMV, but that's not why we teach children to read. We teach them to read for the higher purpose of allowing them access to beautiful and meaningful ideas.
Paul Lockhart, *Lockhart's Lament*

Although this book will touch on on some exciting applications of computing, our primary focus is on the second reason, which may seem more surprising. Computing changes how we think about problems and how we understand the world. The goal of this book is to teach you that new way of thinking.

1.1 Processes, Procedures, and Computers

information processes

Computer science is the study of *information processes*. A process is a sequence of steps. Each step changes the state of the world in some small way, and the result of all the steps produces some goal state. For example, baking a cake, mailing a letter, and planting a tree are all processes. Because they involve physical things like sugar and dirt, however, they are not pure information processes. Computer science focuses on processes that involve abstract information rather than physical things.

The boundaries between the physical world and pure information processes, however, are often fuzzy. Real computers operate in the physical world: they obtain input through physical means (e.g., a user pressing a key on a keyboard that produces an electrical impulse), and produce physical outputs (e.g., an image displayed on a screen). By focusing on abstract information, instead of the physical ways of representing and manipulating information, we simplify computation to its essence to better enable understanding and reasoning.

procedure

A *procedure* is a description of a process. A simple process can be described just by listing the steps. The list of steps is the procedure; the act of following them is the process. If the description can be followed without any thought, we call it a *mechanical procedure*. An *algorithm* is a procedure that is guaranteed to always finish.

algorithm

A mathematician is a machine for turning coffee into theorems.
Attributed to Paul Erdös

For example, here is a procedure for making coffee, adapted from the actual directions that come with a major coffeemaker:

1. Lift and open the coffeemaker lid.
2. Place a basket-type filter into the filter basket.
3. Add the desired amount of coffee and shake to level the coffee.
4. Fill the decanter with cold, fresh water to the desired capacity.
5. Pour the water into the water reservoir.
6. Close the lid.
7. Place the empty decanter on the warming plate.
8. Press the ON button.

Describing processes by just listing steps like this has many limitations. First, natural languages are very imprecise and ambiguous. The steps described rely on the operator knowing lots of unstated assumptions. For example, step three assumes the reader understands the difference between coffee grounds and drinkable coffee, and can correctly infer that this use of "coffee" refers to

coffee grounds. Other steps assume the coffeemaker is plugged in to a power outlet and sitting on a flat surface.

One could, of course, add lots more details to our procedure and make the language more precise than this. Even when a lot of effort is put into writing precisely and clearly, however, natural languages such as English are inherently ambiguous. This is why the United States tax code is 3.4 million words long, but lawyers can still spend years arguing over what it really means.

If you steal property, you must report its fair market value in your income in the year you steal it unless in the same year, you return it to its rightful owner.
Your Federal Income Tax, IRS Publication 17, p. 90.

Another problem with this way of describing a procedure is that the size of the description is proportional to the number of steps in the process. This is fine for simple processes that can be executed by humans in a reasonable amount of time, but the processes we want to execute on computers involve trillions of steps. This means we need more efficient ways to describe them than just listing each step one-by-one. The languages we use to program computers provide ways to define long and complex processes with short procedures.

To program computers, we need tools that allow us to describe processes precisely and succinctly. Since the procedures are carried out by a machine, every step needs to be described; we cannot rely on the operator having "common sense" (for example, to know how to fill the coffeemaker with water without explaining that water comes from a faucet, and how to turn the faucet on). Instead, we need mechanical procedures that can be followed without any thinking.

A *computer* is a machine that can: *computer*

1. Accept input. Input could be entered by a human typing at a keyboard, received over a network, or provided automatically by sensors attached to the computer.
2. Execute a mechanical procedure, that is, a procedure where each step can be executed without any thought.
3. Produce output. Output could be data displayed to a human, but it could also be anything that effects the world outside the computer such as electrical signals that control how a device operates.

Computers exist in a wide range of forms, and thousands of computers are hidden in devices we use everyday but don't think of as computers such as cars, phones, TVs, microwave ovens, and access cards. Our primary focus in this book is on *universal computers*, which are computers that can perform *all* possible mechanical computations on discrete inputs except for practical limits on space and time. The next section explains what it discrete inputs means; Chapters 6 and 12 explore more precisely what it means for a computer to be universal. *universal computers*

1.2 Measuring Computing Power

For physical machines, we can compare the power of different machines by measuring the amount of mechanical work they can perform within a given

amount of time. This power can be captured with units like *horsepower* and *watt*. Physical power is not a very useful measure of computing power, though, since the amount of computing achieved for the same amount of energy varies greatly. Energy is consumed when a computer operates, but consuming energy is not the purpose of using a computer.

The two main properties we can measure about the power of a computing machine are:

1. *How much information* it can process?
2. *How fast* can it process?

We will defer considering the second property until later (starting with Chapter 7), but consider the first question here.

1.2.1 Information

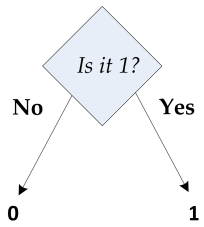
information Informally, we use *information* to mean knowledge. But to understand information quantitatively, as something we can measure, we need a more precise way to think about information.

The way computer scientists measure information is based on how what is known changes as a result of obtaining the information. The primary unit of
bit information is a *bit*. *One bit* of information *halves* the amount of uncertainty. It is equivalent to answering a "yes" or "no" question, where either answer is equally likely beforehand. Before learning the answer, there were two possibilities; after learning the answer, there is one.

binary question We call a question with two possible answers a *binary question*. Since a bit can have two possible values, we often represent the values as **0** and 1.

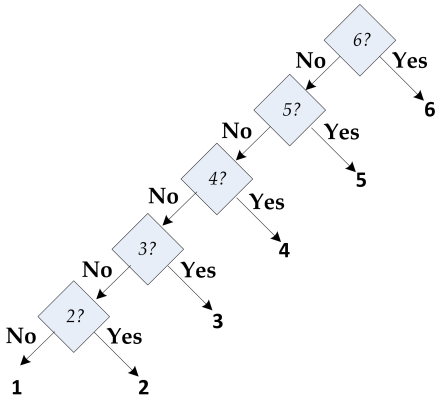
For example, suppose we perform a fair coin toss but do not reveal the result. Half of the time, the coin will land "heads", and the other half of the time the coin will land "tails". Without knowing any more information, our chances of guessing the correct answer are $\frac{1}{2}$. One bit of information would be enough to convey either "heads" or "tails"; we can use **0** to represent "heads" and 1 to represent "tails". So, the amount of information in a coin toss is one bit.

Similarly, one bit can distinguish between the values 0 and 1:



Example 1.1: Dice. How many bits of information are there in the outcome of tossing a fair six-sided die?

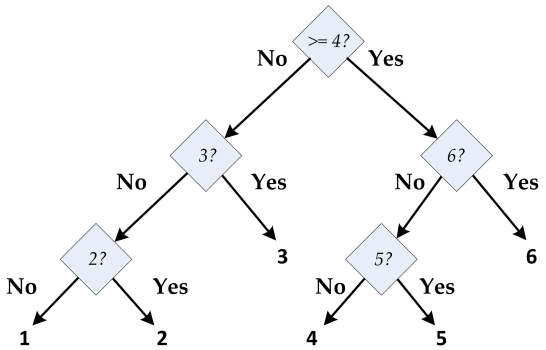
There are six equally likely possible outcomes, so without any more information we have a one in six chance of guessing the correct value. One bit is not enough to identify the actual number, since one bit can only distinguish between two values. We could use five binary questions like this:



This is quite inefficient, though, since we need up to five questions to identify the value (and on average, expect to need $3\frac{1}{3}$ questions.

Can we identify the value with fewer than 5 questions?

Our goal is to identify questions where the "yes" and "no" answers are equally likely—that way, each answer provides the most information possible. This is not the case if we start with, "Is the value 6?", since that answer is expected to be "yes" only one time in six. Instead, we should start with a question like, "Is the value at least 4?". Here, we expect the answer to be "yes" one half of the time, and the "yes" and "no" answers are equally likely. If the answer is "yes", we know the result is 4, 5, or 6. With two more bits, we can distinguish between these three values (note that two bits is actually enough to distinguish among *four* different values, so some information is wasted here). Similarly, if the answer to the first question is no, we know the result is 1, 2, or 3. We need two more bits to distinguish which of the three values it is. Thus, with three bits, we can distinguish all six possible outcomes.



Three bits can convey more information that just six possible outcomes, how-

ever. In the binary question tree, there are some questions where the answer is not equally likely to be "yes" and "no" (for example, we expect the answer to "Is the value 3?" to be "yes" only one out of three times). Hence, we are not obtaining a full bit of information with each question.

Each bit doubles the number of possibilities we can distinguish, so with three bits we can distinguish between $2 * 2 * 2 = 8$ possibilities. In general, with n bits, we can distinguish between 2^n possibilities. Conversely, distinguishing among k possible values requires $\log_2 k$ bits. The *logarithm* is defined such that if $a = b^c$ then $\log_b a = c$. Since each bit has two possibilities, we use the logarithm base 2 to determine the number of bits needed to distinguish among a set of distinct possibilities. For our six-sided die, $\log_2 6 \approx 2.58$, so we need approximately 2.58 binary questions. But, questions are discrete: we can't ask 0.58 of a question, so we need to use three binary questions.

logarithm

Trees. Figure 1.1 depicts a structure of binary questions for distinguishing among eight values. We call this structure a *binary tree*. We will see many useful applications of tree-like structures in this book.

binary tree

Computer scientists draw trees upside down. The *root* is the top of the tree, and the *leaves* are the numbers at the bottom (0, 1, 2, . . ., 7). There is a unique path from the root of the tree to each leaf. Thus, we can describe each of the eight possible values using the answers to the questions down the tree. For example, if the answers are "No", "No", and "No", we reach the leaf 0; if the answers are "Yes", "No", "Yes", we reach the leaf 5.

We can describe any non-negative integer using bits in this way, by just adding additional levels to the tree. For example, if we wanted to distinguish between 16 possible numbers, we would add a new question, "Is is $>= 8$?" to the top of the tree. If the answer is "No", we use the tree in Figure 1.1 to distinguish numbers between 0 and 7. If the answer is "Yes", we use a tree similar to the one in Figure 1.1, but add 8 to each of the numbers in the questions and the leaves.

The *depth* of a tree is the length of the longest path from the root to any leaf. The example tree has depth three. A binary tree of depth d can distinguish up to 2^d different values.

depth

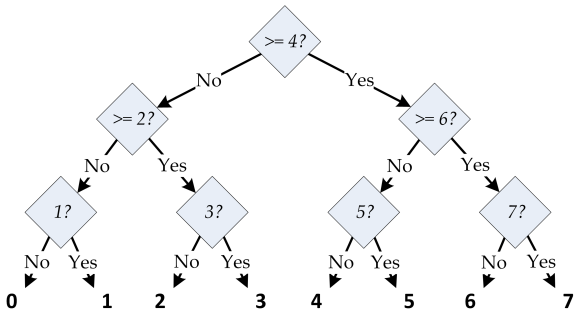


Figure 1.1. Using three bits to distinguish eight possible values.

Units of Information. One *byte* is defined as eight bits. Hence, one byte of information corresponds to eight binary questions, and can distinguish among 2^8 (256) different values. For larger amounts of information, we use metric prefixes, but instead of scaling by factors of 1000 they scale by factors of 2^{10} (1024). Hence, one *kilobyte* is 1024 bytes; one *megabyte* is 2^{20} (approximately one million) bytes; one *gigabyte* is 2^{30} (approximately one billion) bytes; and one *terabyte* is 2^{40} (approximately one trillion) bytes.

Exercise 1.1. Draw a binary tree for distinguishing among the sixteen numbers $0, 1, 2, \ldots, 15$ with the minimum possible depth.

Exercise 1.2. Draw a binary tree for distinguishing among the twelve months of the year with the minimum possible depth.

Exercise 1.3. How many bits are needed:

a. To uniquely identify any currently living human?

b. To uniquely identify any human who ever lived?

c. To identify any location on Earth within one square centimeter?

d. To uniquely identify any atom in the observable universe?

Exercise 1.4. The examples all use binary questions for which there are two possible answers. Suppose instead of basing our decisions on bits, we based it on *trits* where one trit can distinguish between three equally likely values. For each trit, we can ask a ternary question (a question with three possible answers).

a. How many trits are needed to distinguish among eight possible values? (A convincing answer would show a ternary tree with the questions and answers for each node, and argue why it is not possible to distinguish all the values with a tree of lesser depth.)

b. [⋆] Devise a general formula for converting between bits and trits. How many trits does it require to describe b bits of information?

Exploration 1.1: Guessing Numbers

The guess-a-number game starts with one player (the *chooser*) picking a number between 1 and 100 (inclusive) and secretly writing it down. The other player (the *guesser*) attempts to guess the number. After each guess, the chooser responds with "correct" (the guesser guessed the number and the game is over), "higher" (the actual number is higher than the guess), or "lower" (the actual number is lower than the guess).

a. Explain why the guesser can receive slightly more than one bit of informa-

tion for each response.

b. Assuming the chooser picks the number randomly (that is, all values between 1 and 100 are equally likely), what are the best first guesses? Explain why these guesses are better than any other guess. (Hint: there are two equally good first guesses.)

c. What is the maximum number of guesses the second player should need to always find the number?

d. What is the average number of guesses needed (assuming the chooser picks the number randomly as before)?

e. [⋆] Suppose instead of picking randomly, the chooser picks the number with the goal of maximizing the number of guesses the second player will need. What number should she pick?

f. [⋆⋆] How should the guesser adjust her strategy if she knows the chooser is picking adversarially?

g. [⋆⋆] What are the best strategies for both players in the adversarial guess-a-number game where chooser's goal is to pick a starting number that maximizes the number of guesses the guesser needs, and the guesser's goal is to guess the number using as few guesses as possible.

20Q Game

Image from ThinkGeek

Exploration 1.2: Twenty Questions

The two-player game *twenty questions* starts with the first player (the *answerer*) thinking of an object, and declaring if the object is an animal, vegetable, or mineral (meant to include all non-living things). After this, the second player (the *questioner*), asks binary questions to try and guess the object the first player thought of. The first player answers each question "yes" or "no". The website http://www.20q.net/ offers a web-based twenty questions game where a human acts as the answerer and the computer as the questioner. The game is also sold as a $10 stand-alone toy (shown in the picture).

a. How many different objects can be distinguished by a perfect questioner for the standard twenty questions game?

b. What does it mean for the questioner to play perfectly?

c. Try playing the 20Q game at http://www.20q.net. Did the computer guess your item?

d. Instead of just "yes" and "no", the 20Q game offers four different answers: "Yes", "No", "Sometimes", and "Unknown". (The website version of the game also has "Probably", "Irrelevant", and "Doubtful".) If all four answers were equally likely (and meaningful), how many items could be distinguished in 20 questions?

e. For an Animal, the first question 20Q asks is "Does it jump?" (note that 20Q will select randomly among a few different first questions). Is this a good first question?

f. [⋆] How many items do you think 20Q has data for?

g. [⋆⋆] Speculate on how 20Q could build up its database.

1.2.2 Representing Data

We can use sequences of bits to represent many kinds of data. All we need to do is think of the right binary questions for which the bits give answers that allow us to represent each possible value. Next, we provide examples showing how bits can be used to represent numbers, poems, and pictures.

Numbers. In the previous section, we saw how to distinguish a set of items using a tree where each node asks a binary question, and the branches correspond to the "Yes" and "No" answers. A more compact way of writing down our decisions following the tree is to use 0 to encode a "No" answer, and 1 to encode a "Yes" answer.

We can describe a path to a leaf by a sequence of 0s and 1s—the "No", "No", "No" path to 0 is encoded as 000, and the "Yes", "No", "Yes" path to 5 is encoded as 101. This is known as the *binary number system*. Whereas the decimal number system uses ten as its base (there are ten decimal digits, and the positional values increase as powers of ten), the binary system uses two as its base (there are two binary digits, and the positional values increase as powers of two).

binary number system

For example, the binary number 10010110 represents the decimal value 150. As in the decimal number system, the value of each binary digit depends on its position:

Binary:	1	0	0	1	0	1	1	0
Value:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Value:	128	64	32	16	8	4	2	1

There are only 10 types of people in the world: those who understand binary, and those who don't.
Infamous T-Shirt

By using more bits, we can represent larger numbers. With enough bits, we can represent any natural number this way. The more bits we have, the larger the set of possible numbers we can represent. As we saw with the binary decision trees, n bits can be used to represent 2^n different numbers.

Discrete Values. We can use a finite sequence of bits to describe *any* value that is selected from a *countable* set of possible values. A set is *countable* if there is a way to assign a unique natural number to each element of the set. All finite sets are countable. Some, but not all, infinite sets are countable. For example, there appear to be more integers than there are natural numbers since for each natural number, n, there are two corresponding integers, n and $-n$. But, the integers are in fact countable. We can enumerate the integers as: $0, 1, -1, 2, -2, 3, -3, 4, -4, \ldots$ and assign a unique natural number to each integer in turn.

countable

Other sets, such as the real numbers, are uncountable. Georg Cantor proved this using a technique known as *diagonalization*. Suppose the real numbers are enumerable. This means we could list all the real numbers in order, so we could assign a unique integer to each number. For example, considering just the real numbers between 0 and 1, our enumeration might be:

diagonalization

1	.00000000000000 . . .
2	.25000000000000 . . .
3	.33333333333333 . . .
4	.66666666666666 . . .
. . .	. . .
57236	.141592653589793 . . .
. . .	. . .

Cantor proved by contradiction that there is no way to enumerate all the real numbers. The trick is to produce a new real number that is not part of the enumeration. We can do this by constructing a number whose first digit is different from the first digit of the first number, whose second digit is different from the second digit of the second number, etc. For the example enumeration above, we might choose .1468 . . ..

The k^{th} digit of the constructed number is different from the k^{th} digit of the number k in the enumeration. Since the constructed number differs in at least one digit from every enumerated number, it does not match any of the enumerated numbers exactly. Thus, there is a real number that is not included in the enumeration list, and it is impossible to enumerate all the real numbers.

The property that there are more real numbers than natural numbers has important implications for what can and cannot be computed, which we return to in Chapter 12. For now, the important point is that computers can operate on any inputs that are discrete values. Continuous values, such as real numbers, can only be approximated by computers. Next, we consider how two types of data, text and images, can be represented by computers. The first type, text, is discrete and can be represented exactly; images are continuous, and can only be represented approximately.

Text. The set of all possible sequences of characters is countable. One way to see this is to observe that we could give each possible text fragment a unique number, and then use that number to identify the item. For example we could enumerate all texts alphabetically by length (here, we limit the characters to lowercase letters):

a, b, c, . . ., z, aa, ab, . . ., az, ba, . . ., zz, aaa, . . .

Since we have seen that we can represent all the natural numbers with a sequence of bits, so once we have the mapping between each item in the set and a unique natural number, we can represent all of the items in the set. For the representation to be useful, though, we usually need a way to construct the corresponding number for any item directly.

Instead of enumerating a mapping between all possible character sequences and the natural numbers we need a process for converting any text to a unique number that represents that text. Suppose we limit our text to characters in the standard English alphabet. If we include lower-case letters (26), upper-case letters (26), and punctuation (space, comma, period, newline, semi-colon), we have 57 different symbols to represent. We can assign a unique number to

each symbol, and encode the corresponding number with six bits (this leaves seven values unused since six bits can distinguish 64 values). For example, we could encode using the mapping shown in Table 1.1. The first bit answers the question: "Is it an uppercase letter after **F** or a special character?". When the first bit is 0, the second bit answers the question: "Is it after **p**?".

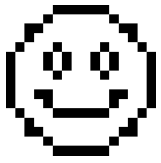
a	**000000**	G	**100000**
b	**000001**	H	**100001**
c	**000010**	...	...
d	**000011**	Z	**110011**
...	...	*space*	**110100**
p	**001111**	,	**110101**
q	**010000**	.	**110110**
...	...	*newline*	**110111**
z	**011001**	;	**111000**
A	**011010**	unused	**111001**
...	...	...	...
F	**011111**	unused	**111111**

Table 1.1. Encoding characters using bits.

This encoding is not the one typically used by computers. One commonly used encoding known as ASCII (the American Standard Code for Information Interchange) uses seven bits so that 128 different symbols can be encoded. The extra symbols are used to encode more special characters.

Once we have a way of mapping each individual letter to a fixed-length bit sequence, we could write down any poem by just concatenating the bits encoding each letter. So, "The" would be encoded as 101101000111000100. We could write down text of length n that is written in the 57-symbol alphabet using this encoding using $6n$ bits. To convert the number back into text, we just need to invert the mapping, replacing each group of six bits with the corresponding letter.

Rich Data. We can use bit sequences to represent complex data like pictures, movies, and audio recordings too. Consider a simple black and white picture:



Since the picture is divided into discrete squares known as *pixel*s, we could encode this as a sequence of bits by using one bit to encode the color of each pixel (for example, using 1 to represent black, and 0 to represent white). This image is 16x16, so has 256 pixels total. We could represent the image using a sequence of 256 bits (starting from the top left corner):

pixel

```
0000011111100000
0000100000010000
0011000000001100
0010000000000100
   ...
```

What about complex pictures that are not divided into discrete squares or a fixed number of colors, like Van Gogh's *Starry Night*?

Different wavelengths of electromagnetic radiation have different colors. For example, light with wavelengths between 625 and 730 nanometers appears red. But, each wavelength of light has a slightly different color; for example, light with wavelength 650 nanometers would be a different color (albeit imperceptible to humans) from light of wavelength 650.0000001 nanometers. There are arguably infinitely many different colors, corresponding to different wavelengths of visible light.[1] Since the colors are continuous and not discrete, there is no way to map each color to a unique, finite bit sequence.

On the other hand, the human eye and brain have limits. We cannot actually perceive infinitely many different colors; at some point the wavelengths are too close for us to distinguish. Ability to distinguish colors varies, but most humans can perceive only a few million different colors. The set of colors that can be distinguished by a typical human is finite; any finite set is countable, so we can map each distinguishable color to a unique bit sequence.

A common way to represent color is to break it into its three primary components (red, green, and blue), and record the intensity of each component. The more bits available to represent a color, the more different colors that can be represented.

[1] Whether there are actually infinitely many different colors comes down to the question of whether the space-time of the universe is continuous or discrete. Certainly in our common perception it seems to be continuous—we can imagine dividing any length into two shorter lengths. In reality, this may not be the case at extremely tiny scales. It is not known if time can continue to be subdivided below 10^{-40} of a second.

Thus, we can represent a picture by recording the approximate color at each point. If space in the universe is continuous, there are infinitely many points. But, as with color, once the points get smaller than a certain size they are imperceptible. We can approximate the picture by dividing the canvas into small regions and sampling the average color of each region. The smaller the sample regions, the more bits we will have and the more detail that will be visible in the image. With enough bits to represent color, and enough sample points, we can represent any image as a sequence of bits.

Summary. We can use sequences of bits to represent any natural number exactly, and hence, represent any member of a countable set using a sequence of bits. The more bits we use the more different values that can be represented; with n bits we can represent 2^n different values.

We can also use sequences of bits to represent rich data like images, audio, and video. Since the world we are trying to represent is continuous there are infinitely many possible values, and we cannot represent these objects exactly with any finite sequence of bits. However, since human perception is limited, with enough bits we can represent any of these adequately well. Finding ways to represent data that are both efficient and easy to manipulate and interpret is a constant challenge in computing. Manipulating sequences of bits is awkward, so we need ways of thinking about bit-level representations of data at higher levels of abstraction. Chapter 5 focuses on ways to manage complex data.

1.2.3 Growth of Computing Power

The number of bits a computer can store gives an upper limit on the amount of information it can process. Looking at the number of bits different computers can store over time gives us a rough indication of how computing power has increased. Here, we consider two machines: the Apollo Guidance Computer and a modern laptop.

The Apollo Guidance Computer was developed in the early 1960s to control the flight systems of the Apollo spacecraft. It might be considered the first *personal computer*, since it was designed to be used in real-time by a single operator (an astronaut in the Apollo capsule). Most earlier computers required a full room, and were far too expensive to be devoted to a single user; instead, they processed jobs submitted by many users in turn. Since the Apollo Guidance Computer was designed to fit in the Apollo capsule, it needed to be small and light. Its volume was about a cubic foot and it weighed 70 pounds. The AGC was the first computer built using integrated circuits, miniature electronic circuits that can perform simple logical operations such as performing the logical *and* of two values. The AGC used about 4000 integrated circuits, each one being able to perform a single logical operation and costing $1000. The AGC consumed a significant fraction of all integrated circuits produced in the mid-1960s, and the project spurred the growth of the integrated circuit industry.

Apollo Guidance Computer

The AGC had 552 960 bits of memory (of which only 61 440 bits were modifi-

able, the rest were fixed). The smallest USB flash memory you can buy today (from SanDisk in December 2008) is the 1 gigabyte Cruzer for $9.99; 1 gigabyte (GB) is 2^{30} bytes or approximately 8.6 billion bits, about 140 000 times the amount of memory in the AGC (and all of the Cruzer memory is modifiable). A typical low-end laptop today has 2 gigabytes of RAM (fast memory close to the processor that loses its state when the machine is turned off) and 250 gigabytes of hard disk memory (slow memory that persists when the machine is turned off); for under $600 today we get a computer with over 4 million times the amount of memory the AGC had.

Moore's law is a violation of Murphy's law. Everything gets better and better.
Gordon Moore

Improving by a factor of 4 million corresponds to doubling 22 times ($2^{22} = 4,194,304$). The amount of computing power approximately doubled every two years between the AGC in the early 1960s and a modern laptop today (2009). This property of exponential improvement in computing power is known as *Moore's Law*. Gordon Moore, a co-founder of Intel, observed in 1965 than the number of components that can be built in integrated circuits for the same cost was approximately doubling every year (revisions to Moore's observation have put the doubling rate at approximately 18 months instead of one year). This progress has been driven by the growth of the computing industry, increasing the resources available for designing integrated circuits. Another driver is that today's technology is used to design the next technology generation. Improvement in computing power has followed this exponential growth remarkably closely over the past 40 years, although there is no law that this growth must continue forever.

Although our comparison between the AGC and a modern laptop shows an impressive factor of 4 million improvement, it is much slower than Moore's law would suggest. Instead of 22 doublings in power since 1963, there should have been 30 doublings (using the 18 month doubling rate). This would produce an improvement of one billion times instead of just 4 million. The reason is our comparison is very unequal relative to cost: the AGC was the world's most expensive small computer of its time, reflecting many millions of dollars of government funding. Computing power available for similar funding today is well over a billion times more powerful than the AGC.

1.3 Science, Engineering, and Liberal Art

Much ink and many bits have been spent debating whether computer science is an art, an engineering discipline, or a science. The confusion stems from the nature of computing as a new field that does not fit well into existing silos. In fact, computer science fits into all three kingdoms, and it is useful to approach computing from all three perspectives.

Science. Traditional science is about understanding nature through observation. The goal of science is to develop general and predictive theories that allow us to understand aspects of nature deeply enough to make accurate quantitative predications. For example, Newton's law of universal gravitation makes predictions about how masses will move. The more general a theory is

the better. A key, as yet unachieved, goal of science is to find a universal law that can describe all physical behavior at scales from the smallest subparticle to the entire universe, and all the bosons, muons, dark matter, black holes, and galaxies in between. Science deals with real things (like bowling balls, planets, and electrons) and attempts to make progress toward theories that predict increasingly precisely how these real things will behave in different situations.

Computer science focuses on artificial things like numbers, graphs, functions, and lists. Instead of dealing with physical things in the real world, computer science concerns abstract things in a virtual world. The numbers we use in computations often represent properties of physical things in the real world, and with enough bits we can model real things with arbitrary precision. But, since our focus is on abstract, artificial things rather than physical things, computer science is not a traditional natural science but a more abstract field like mathematics. Like mathematics, computing is an essential tool for modern science, but when we study computing on artificial things it is not a natural science itself.

In a deeper sense, computing pervades all of nature. A long term goal of computer science is to develop theories that explain how nature computes. One example of computing in nature comes from biology. Complex life exists because nature can perform sophisticated computing. People sometimes describe DNA as a "blueprint", but it is really much better thought of as a program. Whereas a blueprint describes what a building should be when it is finished, giving the dimensions of walls and how they fit together, the DNA of an organism encodes a process for growing that organism. A human genome is not a blueprint that describes the body plan of a human, it is a program that turns a single cell into a complex human given the appropriate environment. The process of evolution (which itself is an information process) produces new programs, and hence new species, through the process of natural selection on mutated DNA sequences. Understanding how both these processes work is one of the most interesting and important open scientific questions, and it involves deep questions in computer science, as well as biology, chemistry, and physics.

The questions we consider in this book focus on the question of what can and cannot be computed. This is both a theoretical question (what can be computed by a given theoretical model of a computer, the focus of Chapter 12), and a pragmatic one (what can be computed by physical things in our universe, the focus of Chapter 13).

Scientists study the world as it is; engineers create the world that never has been.
Theodore von Kármán

Engineering. Engineering is about making useful things. Engineering is often distinguished from crafts in that engineers use scientific principles to create their designs, and focus on designing under practical constraints. As William Wulf and George Fisher put it:[2]

> *Whereas science is analytic in that it strives to understand nature, or what is, engineering is synthetic in that it strives to create. Our own*

[2] William Wulf and George Fisher, A Makeover for Engineering Education, *Issues in Science and Technology*, Spring 2002 (http://www.issues.org/18.3/p_wulf.html).

favorite description of what engineers do is "design under constraint". Engineering is creativity constrained by nature, by cost, by concerns of safety, environmental impact, ergonomics, reliability, manufacturability, maintainability–the whole long list of such "ilities". To be sure, the realities of nature is one of the constraint sets we work under, but it is far from the only one, it is seldom the hardest one, and almost never the limiting one.

Computer scientists do not face the natural constraints faced by civil and mechanical engineers—computer programs are massless, odorless, and tasteless, so the kinds of physical constraints like gravity that impose limits on bridge designs are not relevant to most computer scientists. As we saw from the Apollo Guidance Computer comparison, practical constraints on computing power change rapidly — the one billion times improvement in computing power is unlike any change in physical materials[3]. Although we may need to worry about manufacturability and maintainability of storage media (such as the disk we use to store a program), our focus as computer scientists is on the abstract bits themselves, not how they are stored.

Computer scientists, however, do face many constraints. A primary constraint is the capacity of the human mind—there is a limit to how much information a human can keep in mind at one time. As computing systems get more complex, there is no way for a human to understand the entire system at once. To build complex systems, we need techniques for managing complexity. The primary tool computer scientists use to manage complexity is *abstraction*. Abstraction is a way of giving a name to something in a way that allows us to hide unnecessary details. By using carefully designed abstractions, we can construct complex systems with reliable properties while limiting the amount of information a human designer needs to keep in mind at any one time.

abstraction

Liberal Art. The notion of the *liberal arts* emerged during the middle ages to distinguish education for the purpose of expanding the intellects of free people from the *illiberal arts* such as medicine and carpentry that were pursued for economic purposes. The liberal arts were intended for people who did not need to learn an art to make a living, but instead had the luxury to pursue purely intellectual activities for their own sake. The traditional seven liberal arts started with the *Trivium* (three roads), focused on language:[4]

I must study politics and war that my sons may have liberty to study mathematics and philosophy. My sons ought to study mathematics and philosophy, geography, natural history, naval architecture, navigation, commerce, and agriculture, in order to give their children a right to study painting, poetry, music, architecture, statuary, tapestry, and porcelain.
John Adams, 1780

- Grammar — "the art of inventing symbols and combining them to express thought"
- Rhetoric — "the art of communicating thought from one mind to another, the adaptation of language to circumstance"
- Logic — "the art of thinking"

The Trivium was followed by the *Quadrivium*, focused on numbers:

[3] For example, the highest strength density material available today, carbon nanotubes, are perhaps 300 times stronger than the best material available 50 years ago.

[4] The quotes defining each liberal art are from Miriam Joseph (edited by Marguerite McGlinn), *The Trivium: The Liberal Arts of Logic, Grammar, and Rhetoric*, Paul Dry Books, 2002.

- Arithmetic — "theory of number"
- Geometry — "theory of space"
- Music — "application of the theory of number"
- Astronomy — "application of the theory of space"

All of these have strong connections to computer science, and we will touch on each of them to some degree in this book.

Language is essential to computing since we use the tools of language to describe information processes. The next chapter discusses the structure of language and throughout this book we consider how to efficiently use and combine symbols to express meanings. Rhetoric encompasses communicating thoughts between minds. In computing, we are not typically communicating directly between minds, but we see many forms of communication between entities: interfaces between components of a program, as well as protocols used to enable multiple computing systems to communicate (for example, the HTTP protocol defines how a web browser and web server interact), and communication between computer programs and human users. The primary tool for understanding what computer programs mean, and hence, for constructing programs with particular meanings, is logic. Hence, the traditional trivium liberal arts of language and logic permeate computer science.

The connections between computing and the quadrivium arts are also pervasive. We have already seen how computers use sequences of bits to represent numbers. Chapter 6 examines how machines can perform basic arithmetic operations. Geometry is essential for computer graphics, and graph theory is also important for computer networking. The harmonic structures in music have strong connections to the recursive definitions introduced in Chapter 4 and recurring throughout this book.[5] Unlike the other six liberal arts, astronomy is not directly connected to computing, but computing is an essential tool for doing modern astronomy.

Although learning about computing qualifies as an illiberal art (that is, it can have substantial economic benefits for those who learn it well), computer science also covers at least six of the traditional seven liberal arts.

1.4 Summary and Roadmap

Computer scientists think about problems differently. When confronted with a problem, a computer scientist does not just attempt to solve it. Instead, computer scientists think about a problem as a mapping between its inputs and desired outputs, develop a systematic sequence of steps for solving the problem for any possible input, and consider how the number of steps required to solve the problem scales as the input size increases.

The rest of this book presents a whirlwind introduction to computer science. We do not cover any topics in great depth, but rather provide a broad picture

[5] See Douglas Hofstadter's *Gödel, Escher, Bach* for lots of interesting examples of connections between computing and music.

of what computer science is, how to think like a computer scientist, and how to solve problems.

Part I: Defining Procedures. Part I focuses on how to define procedures that perform desired computations. The nature of the computer forces solutions to be expressed precisely in a language the computer can interpret. This means a computer scientist needs to understand how languages work and exactly what phrases in a language mean. Natural languages like English are too complex and inexact for this, so we need to invent and use new languages that are simpler, more structured, and less ambiguously defined than natural languages. Chapter 2 focuses on language, and during the course of this book we will use language to precisely describe processes and languages are interpreted.

The computer frees humans from having to actually carry out the steps needed to solve the problem. Without complaint, boredom, or rebellion, it dutifully executes the exact steps the program specifies. And it executes them at a remarkable rate — billions of simple steps in each second on a typical laptop. This changes not just the time it takes to solve a problem, but qualitatively changes the kinds of problems we can solve, and the kinds of solutions worth considering. Problems like sequencing the human genome, simulating the global climate, and making a photomosaic not only could not have been solved without computing, but perhaps could not have even been envisioned. Chapter 3 introduces programming, and Chapter 4 develops some techniques for constructing programs that solve problems. To represent more interesting problems, we need ways to manage more complex data. Chapter 5 concludes Part I by exploring ways to represent data and define procedures that operate on complex data.

Part II: Analyzing Procedures. Part II considers the problem of estimating the cost required to execute a procedure. This requires understanding how machines can compute (Chapter 6), and mathematical tools for reasoning about how cost grows with the size of the inputs to a procedure (Chapter 7). Chapter 8 provides some extended examples that apply these techniques.

Part III: Improving Expressiveness. The techniques from Part I and II are sufficient for describing all computations. Our goal, however, it to be able to define concise, elegant, and efficient procedures for performing desired computations. Part III presents techniques that enable more expressive procedures.

Part IV: The Limits of Computing. We hope that by the end of Part III, readers will feel confident that they could program a computer to do just about anything. In Part IV, we consider the question of what can and cannot be done by a mechanical computer. A large class of interesting problems cannot be solved by any computer, even with unlimited time and space. Chapter 13 introduces the most important open problem in computer science. It concerns the question of whether finding an answer is harder than checking if a given answer is correct; it seems obvious that checking an answer should be easier, but for a very interesting class of problems no one has been able to prove that this is the case.

Themes. Much of the book will revolve around three very powerful ideas that are prevalent throughout computing:

Recursive definitions. A recursive definition define a thing in terms of smaller instances of itself. A simple example is defining your ancestors as (1) your parents, and (2) the ancestors of your ancestors. Recursive definitions can define an infinitely large set with a small description. They also provide a powerful technique for solving problems by breaking a problem into solving a simple instance of the problem and showing how to solve a larger instance of the problem by using a solution to a smaller instance. We use recursive definitions to define infinite languages in Chapter 2, to solve problems in Chapter 4, to build complex data structures in Chapter 5. In later chapters, we see how language interpreters themselves can be defined recursively.

Universality. Computers are distinguished from other machines in that their behavior can be changed by a program. Procedures themselves can be described using just bits, so we can write procedures that process procedures as inputs and that generate procedures as outputs. Considering procedures as data is both a powerful problem solving tool, and a useful way of thinking about the power and fundamental limits of computing. We introduce the use of procedures as inputs and outputs in Chapter 4, see how generated procedures can be packaged with state to model objects in Chapter 10. One of the most fundamental results in computing is that any machine that can perform a few simple operations is powerful enough to perform any computation, and in this deep sense, all mechanical computers are equivalent. We introduce a model of computation in Chapter 6, and reason about the limits of computation in Chapter 12.

Abstraction. Abstraction is a way of hiding details by giving things names. We use abstraction to manage complexity. Good abstractions hide unnecessary details so they can be used to build complex systems without needing to understand all the details of the abstraction at once. We introduce procedural abstraction in Chapter 4, data abstraction in Chapter 5, the digital abstraction in Chapter 6, abstraction using objects in Chapter 10, and many other examples of abstraction throughout this book.

Throughout this book, these three themes will recur recursively, universally, and abstractly as we explore the art and science of how to instruct computing machines to perform useful tasks, reason about the resources needed to execute a particular procedure, and understand the fundamental and practical limits on what computers can do.