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 eventually 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 and distances we can travel by the thousands.

Tools that amplify intellectual abilities are much rarer. While many 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 allowed 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 have the ability to 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 the power of automatic computing 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 coolest, most exciting, and most 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

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 actions by a user (e.g., hitting keys), and produce physical outputs (e.g., displaying an image on a screen). By focusing on abstract information, instead of the physical ways of representing and manipulating information, we can simplify computation to the point where we can understand and reason about it more easily.

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*.

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 gently 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.

There are lots of limitations of describing processes by just listing steps this way. First of all, natural languages are very imprecise and ambiguous. The steps described rely on the operator knowing lots of unstated assumptions such as which way the filter should go in the basket, how much coffee should go in the filter (but isn't our end goal to make coffee? here, we need to know that the directions mean coffee grounds), and that the coffeemaker needs to be plugged in to a power outlet and should be sitting on a flat surface. We 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.

I have no idea what you're talking about when you say "ask".
Bill Gates, deposition in Microsoft anti-trust trial

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 with only a few steps, 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 language we use to program computers enable a few statements to describe many different possible steps, and provide ways for defining repetition and changing the steps produced by a given procedure.

To program computers, we need tools that allow us to describe processes precisely and succinctly. Since the procedures will need to be 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:

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. Recall that a mechanical procedure is defined as a procedure where each step can be executed without any thought.
3. Produce output. Output could be printed 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 except for practical limits on space and time. We will explain what this means more precisely in Chapter 13.

1.2 Computing Power

How can we measure the power of a computing machine?

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, not the desired output.

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 chapters, but consider the first question here.

1.2.1 Information

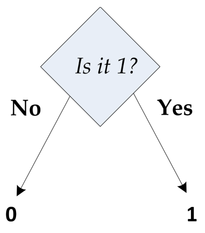
Informally, we use *information* to mean knowledge. But to understand information quantitatively, as something we can measure, we need a more precise way to think about information.

The way computer scientists measure information is based on how the knowledge of some thing changes as a result of obtaining the information.

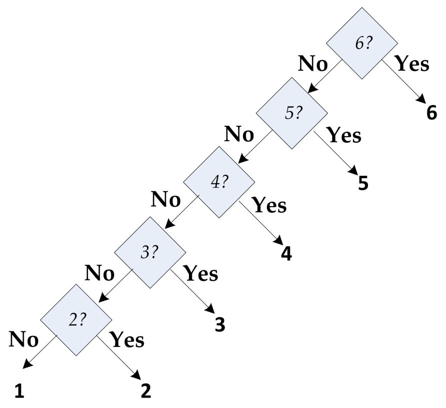
bit The primary unit of information is a *bit*. One bit of information halves the amount of uncertainty. It is equivalent to answering a "yes" or "no" question, where either answer is equally likely beforehand. Before learning the answer, there were two possibilities; after learning the answer, there is one. 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 of information distinguishes between the values 0 and 1:

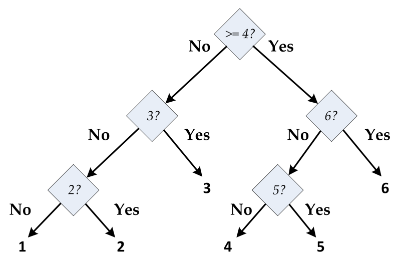


What about rolling a six-sided die? There are six equally likely possible outcomes, so without any more information we have a one in six chance of guessing the correct value. One bit is not enough to identify the actual number, since one bit can only distinguish between two possible values. We could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



This is quite inefficient, though, since in the worst case we need five questions to identify the value. Can we identify the value with fewer than 5 questions?

Our goal is to identify questions where the "yes" and "no" answers are a priori 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, however. From the information tree, we can see 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). This means 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 ≈ 2.58 binary questions. But, questions are discrete: we don’t know how to ask $.58^{th}$ of a question, so we need to use three binary questions.

Figure 1.1 depicts a structure of binary questions for distinguishing among eight values. We call this structure a *tree*; we will see many useful applications of tree-like structures in this book. Computer scientists tend to 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 distinguished 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 the tree is the longest path from the root to any leaf. For the example tree, the depth is three. A binary tree of depth d can distinguish 2^d different values.

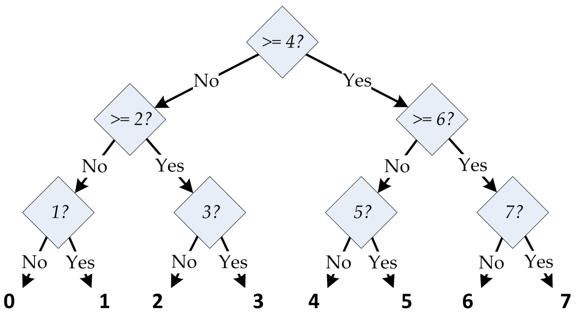


Figure 1.1. Using three bits to distinguish eight possible values.

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.

Excursion 1.1: How Much Data. 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 1.3. 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. That is, how many trits does it require to describe b bits of information?

Excursion 1.2: 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 more than one bit of information 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 antagonistically?

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.

Excursion 1.3: 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 20 questions game where a human acts as the answered and the computer as the questioner. The game is also sold as a $10 stand-alone toy (shown in the picture).

20Q Game

Image from ThinkGeek

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 all 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. Then, we can describe a path to a leaf by a sequence of **0**s and **1**s—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 *Binary Numbers* 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).

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.

Text. We can use a finite sequence of bits to describe *any* value that is selected from a well-defined finite set of possible values. One way to see this is to observe that we could give each item in the set a unique number, and then use that number to identify the item. 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 item in the set. For the representation to be useful, though, we usually need a way to construct the corresponding number for any item directly since maintaining the set of all possible values is not possible.

For example, consider poems. The total number of possible poems is infinite, but for any given length there are a finite (albeit huge) number of possible poems. Enumerating all possible poems of length N is possible in theory, but practically infeasible. Even for haiku, which are short, 3-line poems, this is impossible; there are fewer atoms in the universe than the number of possible 50-letter poems.

So, instead of enumerating a mapping between all possible poems and the natural numbers, we need a process for converting any poem to a unique number that identifies that poem. Suppose we write our poem using the 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 discrete 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 is enough to distinguish 64 values). For example, we could encode the alphabet as:

a	**000000**
b	**000001**
c	**000010**
d	**000011**
...	...
p	**001111**
q	**010000**
...	...
z	**011001**
A	**011010**
...	...
F	**011111**

G	**100000**
H	**100001**
...	...
Z	**110011**
space	**110100**
,	**110101**
.	**110110**
newline	**110111**
;	**111000**
unused	**111001**
...	...
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.

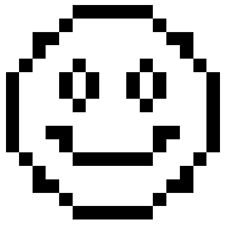
With the encoding in Table 1.1, the first bit answers the question, "Is it an uppercase letter after **F** or a special character?". When the first bit is **0**, the second bit answers the question, "Is it after **p**?".

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, a poem that begins "The" would be encoded as **101101 000111 000100** (the spaces are used for clarity to separate the letters, but are not part of the actual encoding since we know each group of six bits encodes one letter in this encoding). We could write down any poem of length n that is written in the 57-symbol alphabet using this encoding using $6n$ bits. To convert the encoded poem back into English letters, we just need to invert the mapping.

How much information is really in **101101 000111 000100**? It contains 18 bits, so it encodes at most 18 bits of information. But, this is the *maximum* amount of information it could contain. The actual amount of information it encodes, however, is much less. This sequence would contain 18 bits of information only if all sequences of 18 bits are equally likely. In fact, if we already know it is the start of a poem written in English, it encodes far less information. For example, the Wordsworth's complete poetical works contains 887 poems, of which 97 poems start with "The" (that is, the first three letters, so this includes poems like *A Complaint* which starts, "There is a change—and I am poor"). The trigram "The" is the most common starting trigram for Wordsworth's poems, followed by "Whe" (28 poems) and "Wha" (20 poems). Hence, the probability that a poem starts with

"The" is approximately 10% (if Wordsworth is representative of English-language poets), where the probability if all 18-bit sequences were equally likely would be one in $2^{18} = 262,144$. Hence, learning that the poem starts with "The" conveys much less than 18 bits of information, even if our inefficient encoding uses 18 bits to encode the first three letters of the poem.

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



Since the picture is divided into discrete squares (known as *pixels*), 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):

```
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. There are arguably[1] infinitely many different colors, corresponding to different wavelengths of visible light. 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. So, if there are infinitely many colors, there is no way to map each color to a unique, finite bit sequence, and there is no way to completely describe a picture with a 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 close enough that we cannot distinguish them. Ability to distinguish colors varies, but most humans can perceive several million different colors. Then, we can map each distinguishable color to a unique bit sequence. One 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.

So, with bits we can approximate the color at each point. How many different points are there? 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

[1]This comes down to the question of whether the space-time of the universe is really continuous or discrete. Certainly in our common perception it seems to be continuous—we can imagine dividing a time interval in two. In reality, this may not be the case, but if the universe is discrete it is discrete at an extremely tiny scale (e.g., less than 10^{-40} of a second).

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.

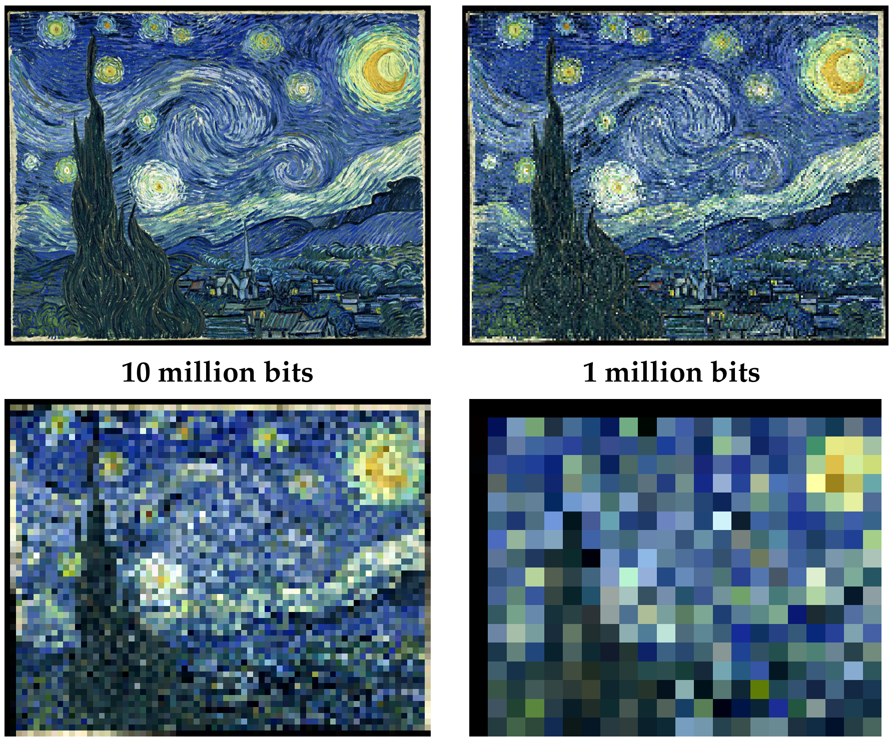


Figure 1.2. Information and image quality.

The first Van Gogh image uses 10 million bits. Figure 1.2 shows four versions of the Van Gogh painting, each using a different number of bits to encode the image. For these images, we use 24-bit color so colors are represented by three 8-bit values, one each for the red, green, and blue intensities. This means 2^{24} (over 16 million) different colors that can be represented.

As with the poems, however, there is much less information in the pictures than the raw number of bits. For the first picture in Figure 1.2, there are approximately 10 million bits—the image is 722x576 pixels, so there are 415 872 pixels; with 24-bits for each pixel, there are 9 980 928 bits (of course, depending on the resolution of your printer or screen, you may not be seeing all the pixels). But, each picture bit does not provide one full bit of information. For example, the color of a pixel is often similar to the color of the pixels next to it. This is not always true, of course, but it is true most of the time. Hence, without learning the color value for this pixel we have a much better that one in 2^{24} chance of guessing this pixel's color. Thus, the actual amount of information we learn from the 24 bits of color information for this pixel are much less than 24 bits.

When images are stored on computers they use *compression* to reduce the number of bits needed to store the image. Compression algorithms can map the 1 million bits in the first image to a smaller number of bits. When the image is viewed, a decompression algorithm performs the inverse mapping, producing the full image from the compressed data. For the first image, one of the most commonly used image compression algorithms (JPEG), reduces the size of the 10 million bit image to just over 1 million bits. JPEG is a *lossy* compression algorithm, meaning that when the compressed image is decompressed it may not produce exactly the original image, but rather a good approximation of it.

compression

Summary and Preview. We can use sequences of bits to represent any natural number exactly, and hence, represent any member of a finite set of values 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; but, since human perception is limited, with enough bits we can represent any of these adequately well. Finding good ways to represent data is a constant challenge in computing. Manipulating sequences of bits is awkward, so we need ways of thinking about sequences of bits 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. In some respects, it can 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.

Apollo Guidance Computer

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. 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.

AGC User Interface

The AGC had 552 960 bits of memory (of which only 61 440 bits were modifiable, the rest were fixed). (Compare the roughly half a million bits needed to perform rendezvous in space with the van Gogh pictures.) 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

To improve by a factor of 4 million involves doubling 23 times in about 46 years. So, 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*. One of the pioneers in integrated circuit technology, Gordon Moore, who co-founded 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 (various revisions to Moore's observation have put the doubling rate at approximately 18 months instead of one year). This growth has been driven by the growth of the computing industry, increasing the resources available for designing integrated circuits. Another driver is that the current technology can be 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, of course, that this growth can 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 23 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 on 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 well into all three categories, 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 is, theories that allow us to understand aspects of nature deeply enough to make accurate quantitative predications about them. 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 particle to the largest galaxy clusters, and all the bosons, muons, dark matter, and black holes in between. Science deals with real things (like bowling balls, planets, and electrons) and attempts to make progress towards theories that predict how these real things will behave in different situations.

Computer science typically focuses on artificial things like numbers, graphs, functions, and lists. Instead of dealing with physical things in the real world, most of computer science concerns abstract things in a virtual world. The numbers we use in computations, of course, can 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 the abstract, artificial things rather than the real, physical things, computer science is traditionally not a 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.

On the other hand, computing exists all over nature. A long term goal of computer science is to develop theories that allow us to better understand how nature computes. One direct example of computing in nature comes from biology. Complex life exists because nature can perform sophisticated computing. Sometimes people analogize 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 needs to encode a process for making that organism. This can be seen most clearly by looking at embryology. In the steps from a single cell to a human baby, the embryo goes through forms that seem to resemble our evolutionary ancestors (see Neil Shubin's *Your Inner Fish* for a wonderful exposition on this).

Surely this is not the most sensible "make-a-human" process if one were designing it from scratch. Thus, the 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. 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 scientific 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), and a pragmatic one (what can be computed by physical things in our universe). Answering the second question requires deeper understanding of our physical universe (in particular, quantum physics) than is currently known, as well as advances in computer science.

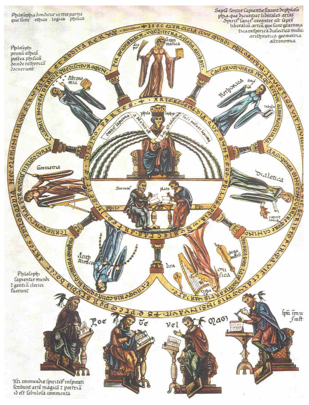
Scientists study the world as it is; engineers create the world that never has been.
Theodore von Kármán

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 (University of Virginia professor and President of the National Academy of Engineering) and George Fisher (CEO of Kodak) wrote:

> *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 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.* [WF02]

Computer scientists do not typically face the main constraints faced by civil and mechanical engineers—computer programs are massless, odorless, and tasteless, so the kinds of physical constraints like gravity and tensile strength that impose limits on bridge designs are not relevant to most computer scientists. As we saw from the Apollo Guidance Computer comparison, what practical constraints there are on computing power change rapidly — the one billion times improvement in computing power is unlike any change in physical materials (for example, the highest strength density material available today, carbon nanotubes, are perhaps 300 times stronger than the best material available 50 years ago). Although we may need to worry about manufacturability and maintainability of storage media (such as the disk we use to store a program), our primary focus as computer scientists is on the abstract bits themselves, not how they are stored.

There are, however, many constraints that computer scientists face. 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.


Herrad von Landsberg, ***Septem Artes Liberales***

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 (the quotes defining each liberal art are from Sister Miriam Joseph [Jos02])

- 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* (four roads), focused on numbers:

- 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. In Chapter 2, we investigate 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 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 humans through their user interfaces. 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 computing uses sequences of bits to represent numbers, and computing requires fast algorithms for performing arithmetic. In Chapter 17, we will see how numbers can be represented using even more fundamental building blocks and arithmetical primitives can be defined starting from scratch. 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 we will see in Chapter 4 and later chapters (see Hofstadter's *Gödel, Escher, Bach* for lots of interesting examples of connections between computing and music). Unlike the other six liberal arts, astronomy is not directly connected to computing, but computing is an essential tool for doing modern astronomy.

Hence, although learning about computing qualifies as an illiberal art (that is, it has 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 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 they mean. The next chapter focuses on language. 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.

The computer frees the human 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 also not have even been envisioned.

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 of what computer science is, how to think like a computer scientist, and how to solve problems. Much of the book will revolve around three very powerful ideas that are prevalent throughout computing:

Recursive definitions. A recursive definition defines something in terms of smaller instances of itself. This is a powerful way of 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 introduce recursive definitions in how they are used to define infinite languages in Chapter 2, show how they can be used to solve problems in Chapter 4, illustrate how complex data structures can be built from recursive definitions in Chapter 5, and in later chapters see how language interpreters themselves can be defined recursively.

Higher order procedures. Computers are distinguished from other machines in that their behavior can be changed by a program. Programs themselves are just bits though, so we can write programs to generate other programs. With higher order procedures, we treat procedures just like any other data. Procedures can be passed as inputs and generated 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 12, and use procedures at data to reason about the fundamental limits of computing in Chapter 13.

Abstraction. Abstraction is a way of hiding details by giving things names. We use abstraction to manage complexity and to enable things to be reused. 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, abstraction using objects in Chapter 12, and many other examples of abstraction throughout this book.

These themes will recur throughout the rest of the book, as we explore the art and science of how to instruct computing machines to perform useful tasks, reasoning about the resources needed to execute a particular procedure, and understanding the fundamental and practical limits on what computers can do.