2
Computation and Representation

“The alphabet was a great invention, which enabled men to store and to learn with little effort what others had learned the hard way – that is, to learn from books rather than from direct, possibly painful, contact with the real world.”, B.F. Skinner

“The name of the song is called ‘HADDOCK’S EYES.’” [said the Knight]
“Oh, that’s the name of the song, is it?” Alice said, trying to feel interested.
“No, you don’t understand,” the Knight said, looking a little vexed. “That’s what the name is CALLED. The name really is ‘THE AGED AGED MAN.’”
“Then I ought to have said ‘That’s what the SONG is called?’” Alice corrected herself.
“No, you oughtn’t: that’s quite another thing! The SONG is called ‘WAYS AND MEANS’: but that’s only what it’s CALLED, you know!”
“Well, what IS the song, then?” said Alice, who was by this time completely bewildered.
“I was coming to that,” the Knight said. “The song really IS ‘A-SITTING ON A GATE’: and the tune’s my own invention.”
Lewis Carroll, Through the looking glass

To a first approximation, computation is a process that maps an input to an output.

When discussing computation, it is essential to separate the question of what is the task we need to perform (i.e., the specification) from the question of how we achieve this task (i.e., the implementation).
For example, as we’ve seen, there is more than one way to achieve the computational task of computing the product of two integers.

Figure 2.1: Our basic notion of computation is some process that maps an input to an output

Learning Objectives:
- Distinguish between specification and implementation, or equivalently between algorithms/programs and mathematical functions.
- See concept of representing an object as a string (often of zeroes and ones).
- Examples of representations for common objects such as numbers, vectors, lists, graphs.
- Prefix-free representations.
- Cantor’s Theorem: The real numbers are cannot be represented exactly as finite strings.
In this chapter we focus on the what part, namely defining computational tasks. For starters, we need to define the inputs and outputs. A priori, capturing all the potential inputs and outputs that we might ever want to compute on seems challenging, since computation today is applied to a wide variety of objects. We do not compute merely on numbers, but also on texts, images, videos, connection graphs of social networks, MRI scans, gene data, and even other programs. We will represent all these objects as strings of zeroes and ones, that is objects such as 0011101 or 1011 or any other finite list of 1’s and 0’s.

Today, we are so used to the notion of digital representation that we are not surprised by the existence of such an encoding. But it is actually a deep insight with significant implications. Many animals can convey a particular fear or desire, but what is unique about humans is language: we use a finite collection of basic symbols to describe a potentially unlimited range of experiences. Language allows transmission of information over both time and space and enables societies that span a great many people and accumulate a body of shared knowledge over time.¹

Over the last several decades, we have seen a revolution in what we can represent and convey in digital form. We can capture experiences with almost perfect fidelity, and disseminate it essentially instantaneously to an unlimited audience. Moreover, once information is in digital form, we can compute over it, and gain insights from data that were not accessible in prior times. At the heart of this revolution is the simple but profound observation that we can represent an unbounded variety of objects using a finite set of symbols (and in fact using only the two symbols 0 and 1).²

In later chapters, we will typically take such representations for granted, and hence use expressions such as “program $P$ takes $x$ as input” when $x$ might be a number, a vector, a graph, or any other object, when we really mean that $P$ takes as input the representation of $x$ as a binary string. However, in this chapter we will dwell a bit more on how we can construct such representations.

### 2.1 DEFINING REPRESENTATIONS

Every time we store numbers, images, sounds, databases, or other objects on a computer, what we actually store in the computer’s memory is the representation of these objects. Moreover, the idea of representation is not restricted to digital computers. When we write down text or make a drawing we are representing ideas or experiences as sequences of symbols (which might as well be strings of zeroes and ones). Even our brain does not store the actual sensory inputs we experience, but rather only our representation of them.

---

¹ For example, at the time I am writing this, the full contents of the English Wikipedia, including all the text and media, can be encoded in a binary string of length $n \sim 10^{12}$ (i.e., about 100 Gigabytes).

² There is nothing “holy” about using zero and one as the basic symbols, and we can (indeed sometimes people do) use any other finite set of two or more symbols as the fundamental “alphabet”. We use zero and one in this course mainly because it simplifies notation.
To use objects such as numbers, images, graphs, or others as inputs for computation, we need to define precisely how to represent these objects as binary strings. A representation scheme is a way to map an object \( x \) to a binary string \( E(x) \in \{0, 1\}^* \). For example, a representation scheme for natural numbers is a function \( E : \mathbb{N} \to \{0, 1\}^* \). Of course, we cannot merely represent all numbers as the string “0011” (for example). A minimal requirement is that if two numbers \( x \) and \( x' \) are different then they would be represented by different strings. Another way to say this, is that we require the encoding function \( E \) to be one to one.

2.1.1 Representing natural numbers.

We now show how we can represent natural numbers as binary strings. Over the years people have represented numbers in a variety of ways, including Roman numerals, tally marks, our own Hindu-Arabic decimal system, and many others. We can use any one of those as well as many others to represent a number as a string.\(^3\) However, for the sake of concreteness, we use the binary basis as our default representation of natural numbers as strings. For example, we represent the number six as the string 110 since \( 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6 \), and similarly we represent the number thirty-five as the string \( y = 100011 \) which satisfies \( \sum_{i=0}^{5} y_i \cdot 2^{|y|-i} = 35 \). Some more examples are given in the table below.

Table 2.1: Representing numbers in the binary basis. The lefthand column contains representations of natural numbers in the decimal basis, while the righthand column contains representations of the same numbers in the binary basis. Note that in both representations the leftmost (i.e., most significant) digit is never equal to zero (unless we represent the natural number zero).

<table>
<thead>
<tr>
<th>Number (decimal representation)</th>
<th>Number (binary representation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>503</td>
<td>111110111</td>
</tr>
<tr>
<td>53</td>
<td>110101</td>
</tr>
<tr>
<td>40</td>
<td>101000</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>40</td>
<td>101000</td>
</tr>
<tr>
<td>801</td>
<td>1100100001</td>
</tr>
<tr>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>389</td>
<td>110000101</td>
</tr>
<tr>
<td>3750</td>
<td>111010100101</td>
</tr>
<tr>
<td>506</td>
<td>111111010</td>
</tr>
</tbody>
</table>

\(^3\) For example, we could represent a number \( x \) by first breaking it apart to its decimal digits, and then represent each digit as a binary string corresponding to its graphical representation (see Fig. 2.3).

\(^4\) We could have equally well reversed the order so as to represent 35 by the string \( y' = 1100011 \) satisfying \( \sum_{i=0}^{5} y'_i \cdot 2^{|y'|-i} = 35 \). Such low level choices will not make a difference in this course. A related (though not identical) distinction is the Big Endian vs. Little Endian representation for integers in computing architectures.

Figure 2.3: Representing each one the digits 0, 1, 2, ..., 9 as a 12 × 8 bitmap image, which can be thought of as a string in \( \{0, 1\}^{96} \). Using this scheme we can represent a natural number \( x \) of \( n \) decimal digits as a string in \( \{0, 1\}^{96n} \). Image taken from blog post of A. C. Andersen.
The binary representation encodes numbers as strings in a one-to-one fashion, and so it yields a proof of the following theorem:

**Theorem 2.1 — Binary representation of natural numbers.** There exists a one-to-one function $NtS : \mathbb{N} \to \{0, 1\}^*$.  

Proof. To prove this theorem, we first precisely define the binary representation function, and then prove that this function is one to one. If $x$ is even then the least significant binary digit of $x$ is zero, while if $x$ is odd, then the least significant binary digit is one. In other words, the least significant binary digit of $x$ is $\text{parity}(x)$ where $\text{parity}(x)$ is defined to be equal 1 if $x$ is odd and defined to equal 0 if $x$ is even. Moreover, for every $x > 1$, the binary representation of the number $\lfloor x/2 \rfloor$ (i.e., the number obtained by “rounding down” $x/2$) is obtained from the binary representation of $x$ by “chopping off” the least significant digit. Hence we can define $NtS$ recursively as follows:

$$NtS(x) = \begin{cases} 
"" & x = 0 \\
NtS(\lfloor x/2 \rfloor)\text{parity}(x) & x > 0 
\end{cases} \quad (2.1)$$

(The function $NtS$ is well defined since for every $x > 0$, $\lfloor x/2 \rfloor < x$.) That is, the binary representation of $x$ is obtained by concatenating the string corresponding binary representation of $\lfloor x/2 \rfloor$ with the single bit $\text{parity}(x)$.

It can be shown (though we omit the proof since it is slightly tedious, and can be easily found in many online resources) that if $y = NtS(x)$ then $x = \sum_{i=0}^{n} y_i \cdot 2^{n-i}$ where $n = |y| − 1$. In particular, this gives a way to recover (or decode) the original $x$ from the output $y = NtS(x)$ which means that $NtS$ is one to one.  

2.1.2 Implementing the binary representation in python (optional)

In the Python programming language, we can compute the above encoding and decoding functions as follows:

```python
from math import floor, log

def NtS(x):
    if x<1: return ""
    return NtS(floor(x/2))+str(x % 2)

print(NtS(236))
# 11101100

print(int2bits(19))
# 10011
```

The representation $NtS$ uses the empty string "" to represent the number 0. However, we can also represent this number using the string 0 as well. This choice will not make any difference for our purposes.
def StN(y):
    x = 0
    n = len(y)-1
    for i in range(n+1):
        x += int(y[i])*(2**(n-i))
    return x

print(StN(NtS(236)))
# 236

We can also implement \( NtS \) non recursively as follows:

def NtS(x):
    def list2string(L):
        return ''.join([str(e) for e in L])
    n = floor(log2(x))  # largest power of two smaller than x
    return list2string([ floor(x / 2**i) % 2 for i in range(n+1)])

Remark 2.2 — Programming examples. In this book, we will often illustrate our points by using programming languages to present certain computations. Our examples will be relatively short, and our point will always be to emphasize that certain computations can be achieved concretely, rather than focusing on any particular language features. We often use Python, but this choice is somewhat arbitrary. Indeed, one of the messages of this course is that all programming languages are in a certain precise sense equivalent to one another, and hence we could have just as well used JavaScript, C, COBOL, Visual Basic or even BrainF*ck. This book is not about programming, and it is absolutely OK if you are not familiar with Python or do not follow the fine details of code examples such as the above. You might still find it instructive to try to parse these examples, with the help of websites such as Google or StackOverflow. In particular, the non-recursive implementation of the function \( NtS \) above uses the fact that the binary representation of a natural number \( x \) is the list \([(\lfloor x / 2^i \rfloor \mod 2)\}_{i=0,...,\lceil \log_2 x \rceil} \), which in Python-speak is written as \([ \text{floor}(x / 2**i) \% 2 \text{ for } i \text{ in range(floor(log2(x))}+1]) \].

2.1.3 Meaning of representations
It is natural for us to think of 236 as the “actual” number, and of 11101100 as “merely” its representation. However, for most Euro-
peans in the middle ages CCXXXVI would be the “actual” number and 236 (if they have even heard about it) would be the weird Hindu-Arabic positional representation.\footnote{While the Babylonians already invented a positional system much earlier, the decimal positional system we use today was invented by Indian mathematicians around the third century. It was taken up by Arab mathematicians in the 8th century. It was mainly introduced to Europe in the 1202 book “Liber Abaci” by Leonardo of Pisa, also known as Fibonacci, but did not displace Roman numerals in common usage until the 15th century.} When our AI robot overlords materialize, they will probably think of 11101100 as the “actual” number and of 236 as “merely” a representation that they need to use when they give commands to humans.

So what is the “actual” number? This is a question that philosophers of mathematics have pondered over throughout history. Plato argued that mathematical objects exist in some ideal sphere of existence (that to a certain extent is more “real” than the world we perceive via our senses, as this latter world is merely the shadow of this ideal sphere). In Plato’s vision, the symbols 236 are merely notation for some ideal object, that, in homage to the late musician, we can refer to as “the number commonly represented by 236”.

The Austrian philosopher Ludwig Wittgenstein, on the other hand, argued that mathematical objects do not exist at all, and the only things that exist are the actual marks on paper that make up 236, 00110111 or CCXXXVI. In Wittgenstein’s view, mathematics is merely about formal manipulation of symbols that do not have any inherent meaning. You can think of the “actual” number as (somewhat recursively) “that thing which is common to 236, 00110111 and CCXXXVI and all other past and future representations that are meant to capture the same object”.

While reading this book, you are free to choose your own philosophy of mathematics, as long as you maintain the distinction between the mathematical objects themselves and the various particular choices of representing them, whether as splatters of ink, pixels on a screen, zeroes and one, or any other form.

\section{2.2 Representing More Objects}

We have seen that natural numbers can be represented as binary strings. We now show that the same is true for other types of objects, including (potentially negative) integers, rational numbers, vectors, lists, graphs and many others. In many instances, choosing the “right” string representation for a piece of data is highly nontrivial, and finding the “best” one (e.g., most compact, best fidelity, most efficiently manipulable, robust to errors, most informative features, etc.) is the object of intense research. But for now, we focus on presenting some simple representations for various objects that we would like to use as inputs and outputs for computation.

\subsection{2.2.1 Representing (potentially negative) integers}

Since we can represent natural numbers as strings, we can represent the full set of integers (i.e., members of the set
\( \mathbb{Z} = \{ \ldots, -3, -2, -1, 0, +1, +2, +3, \ldots \} \) by adding one more bit that represents the sign. If \( x \in \mathbb{Z} \), then we can represent it by the string \( N_t S( |x| ) \sigma \) where \( \sigma \) equals to 0 is \( x \geq 0 \) and \( \sigma \) equals to 1 if \( x < 0 \).

Thus the string \( y \in \{ 0, 1 \}^* \) will represent the number

\[
x = (-1)^{y_{n+1}} \left( \sum_{i=0}^{n-1} y_i \cdot 2^{n-i} \right)
\]

(2.2)

where \( n = |y| - 2 \). Formally, the above can be shown to give a one to one function \( ZtS : \mathbb{Z} \to \{ 0, 1 \}^* \) that maps the integers into strings.

\[\text{Remark 2.3 - Two’s complement representation (optional).}\]

The above approach of representing an integer using a specific “sign bit” is known as the Signed Magnitude Representation and was used in some early computers. However, the two’s complement representation is much more common. The two’s complement representation of an integer \( k \) in the set \( \{-2^n, -2^n + 1, \ldots, 2^n - 1\} \) is the string \( r(k) \) of length \( n + 1 \) defined as follows:

\[
r(k) = \begin{cases} 
    b(k) & 0 \leq k \leq 2^n - 1 \\
    b(2^{n+1} + k) & -2^n \leq k \leq -1 
\end{cases}
\]

(2.3)

where \( b(m) \) denotes the standard binary representation of a number \( m \in \{ 0, \ldots, 2^{n+1} \} \) as string of length \( n + 1 \). (We pad this representation with 0’s to length \( n + 1 \) if needed.) For example, if \( n = 3 \) then \( r(1) = b(1) = 0001 \), \( r(2) = b(2) = 0010 \), \( r(-1) = b(16 - 1) = 1111 \), and \( r(-8) = b(16 - 8) = 1000 \).

If \( k \) is a negative number larger or equal to \( -2^n \) then \( 2^{n+1} + k \) is a number between \( 2^n \) and \( 2^{n+1} - 1 \). Hence the two’s complement representation of such a number \( k \) will be a string of length \( n + 1 \) with its first digit equal to 1.

Another way to say this is that we represent a potentially negative number \( k \in \{-2^n, \ldots, 2^n - 1\} \) as the non-negative number \( k \mod 2^{n+1} \) (see also Fig. 2.4).

\[\text{This means that if two (potentially negative) numbers } k \text{ and } k' \text{ are not too large (i.e., } |k| + |k'| < 2^{n+1} \text{), then we can compute the representation of } k + k' \text{ by adding modulo } 2^{n+1} \text{ the representations of } k \text{ and } k' \text{ as if they were non-negative integers. This property of the two’s complement representation is its main attraction since, depending on their architectures, microprocessors can often perform arithmetic operations modulo } 2^w \text{ very efficiently (for certain values of } w \text{ such as 32 and 64). Many systems leave it to the programmer to check that values are not too large and will carry out this modular arithmetic regardless of the size of the numbers involved. For this reason, in some systems adding two large positive numbers can result in a negative number (e.g., adding}\]

\[\text{adding}\]
If \( k \) is a (potentially negative) integer, and \( m \) is a non-negative number, then \( k \mod m \) is the unique number \( r \in \{0, \ldots, m-1\} \) such that \( k = \ell m + r \) for some \( \ell \in \mathbb{Z} \).

\( 2^n - 100 \) and \( 2^n - 200 \) might result in \(-300\) since \(-300 \mod 2^{n+1} = 2^{n+1} - 300 \), see also Fig. 2.4.

\[ \text{Figure 2.4: In the two's complement representation we represent a potentially negative integer } k \in \{-2^n, \ldots, 2^n - 1\} \text{ as an } n + 1 \text{ length string using the binary representation of the integer } k \mod 2^{n+1}. \text{ On the lefthand side: this representation for } n = 3 \text{ (the red integers are the numbers being represented by the blue binary strings). If a microprocessor does not check for overflows, adding the two positive numbers 6 and 5 might result in the negative number } -5 \text{ (since } -5 \mod 16 = 11 \text{). The righthand side is a C program that will on some 32 bit architecture print a negative number after adding two positive numbers. (Integer overflow in C is considered undefined behavior which means the result of this program, including whether it runs or crashes, could differ depending on the architecture, compiler, and even compiler options and version.)} \]

The decoding function of a representation scheme is always onto since every object must be represented by some string. However, the decoding function is not always one to one. For example, in this particular representation scheme, the two strings 1 and 0 both represent the number zero (since they can be thought of as representing \(-0\) and \(+0\) respectively, can you see why?). We can also allow a partial decoding function for representations. For example, in the representation above there is no number that is represented by the empty string. But this is still a fine representation, since the decoding partial function is onto and the encoding function is the one-to-one total function \( E : \mathbb{Z} \to \{0, 1\}^\ast \) which maps an integer of the form \( a \times k \), where \( a \in \{\pm 1\} \) and \( k \in \mathbb{N} \) to the bit \((-1)^a\) concatenated with the binary representation of \( k \). That is, every integer can be represented as a string, and every two distinct integers have distinct representations.

\[ \text{R} \]

Remark 2.4 — Interpretation and context. Given a string \( y \in \{0, 1\}^\ast \), how do we know if it’s “supposed” to represent a (nonnegative) natural number or a (potentially negative) integer? For that matter, even if we know \( y \) is “supposed” to be an integer, how do we know what representation scheme it uses? The short answer is that we do not necessarily know this information, unless it is supplied from the context. We can treat the same string \( y \) as representing a natural number, an integer, a piece of text, an image, or a green gremlin. Whenever we say a sentence such as “let \( n \) be the number represented by the string \( y \),” we will assume that we are fixing some canonical representation scheme such as the ones above. The choice

---

If \( k \) is a (potentially negative) integer, and \( m \) is a non-negative number, then \( k \mod m \) is the unique number \( r \in \{0, \ldots, m-1\} \) such that \( k = \ell m + r \) for some \( \ell \in \mathbb{Z} \).
of the particular representation scheme will rarely matter, except that we want to make sure to stick with the same one for consistency.

2.2.2 Representing rational numbers

We can represent a rational number of the form \( a/b \) by representing the two numbers \( a \) and \( b \) (again, this is not a unique representation, but this is fine). However, merely concatenating the representations of \( a \) and \( b \) will not work.\(^9\) For example, recall that we represent 4 as 100 and 43 as 101011, but the concatenation 10010111 of these strings is also the concatenation of the representation 10010 of 18 and the representation 1011 of 11. Hence, if we used such simple concatenation then we would not be able to tell if the string 10010111 is supposed to represent 4/43 or 18/11.\(^11\)

The way to tackle this is to find a general representation for pairs of numbers. If we were using a pen and paper, we would just use a separator such as the symbol \( \| \) to represent, for example, the pair consisting of the numbers represented by \((0, 1)\) and \((1, 0, 0, 0, 1)\) as the length-9 string \( s = “01\|110001\” \). Using such separators is similar to the way we use spaces and punctuation to separate words in English. By adding a little redundancy, we can do just that in the digital domain. The idea is that we can map the three element set \( \Sigma = \{0, 1, \|\} \) to the four element set \( \{0, 1\}^2 \) via the one-to-one map that takes 0 to 00, 1 to 11 and \( \| \) to 01.

Example 2.5 — Representing a rational number as a string. Consider the rational number \( r = 19/236 \). In our convention, we represent 19 as the string 10011 and 236 as the string 11101100, and so we could represent \( r \) as the pair of strings \((10011, 11101100)\). We can then represent this pair as the length 14 string 10011\|11101100 over the alphabet \( \{0, 1, \|\} \). Now, applying the map \( 0 \mapsto 00, 1 \mapsto 11, \| \mapsto 01 \), we can represent the latter string as the length 28 string \( s = 11000011101111100111110000 \) over the alphabet \( \{0, 1\} \). So we represent the rational number \( r = 19/36 \) be the binary string \( s = 11000011101111110011111000 \).

More generally, we obtained a representation of the non-negative rational numbers as binary strings by composing the following representations:

1. Representing a non-negative rational number as a pair of natural numbers.
2. Representing a natural number by a string via the binary representation. (We can use the representation of integers to handle rational numbers that can be negative.)

3. Combining 1 and 2 to obtain a representation of a rational number as a pair of strings.

4. Representing a pair of strings over \( \{0, 1\} \) as a single string over \( \Sigma = \{0, 1, \|\} \).

5. Representing a string over \( \Sigma \) as a longer string over \( \{0, 1\} \).

The same idea can be used to represent triples, quadruples, and generally all tuples of strings as a single string (can you see why?). Indeed, this is one instance of a very general principle that we use time and again in both the theory and practice of computer science (for example, in Object Oriented programming):

**Big Idea 1** If we can represent objects of type T as strings, then we can also represent more complex objects built out of T as strings (such as pairs or lists of elements in T, nested lists, and so on and so forth).

We will come back to this point when we discuss prefix free encoding in Section 2.4.2.

### 2.3 Representing Real Numbers

The set of real numbers \( \mathbb{R} \) contains all numbers including positive, negative, and fractional, as well as irrational numbers such as \( \pi \) or \( e \). Every real number can be approximated by a rational number, and thus we can represent every real number \( x \) by a rational number \( a/b \) that is very close to \( x \). For example, we can represent \( \pi \) by \( 22/7 \) within an error of about \( 10^{-3} \). If we want a smaller error (e.g., about \( 10^{-4} \)) then we can use \( 311/99 \), and so on and so forth.

The above representation of real numbers via rational numbers that approximate them is a fine choice for a representation scheme. However, typically in computing applications, it is more common to

![Figure 2.5](image-url)
use the floating point representation scheme to represent real numbers. In the floating point representation scheme we represent $x$ by the pair $(b, e)$ of (positive or negative) integers of some prescribed sizes (determined by the desired accuracy) such that $b \times 2^e$ is closest to $x$ (see Fig. 2.5 for more details). This representation is called “floating point” because we can think of the number $b$ as specifying a sequence of binary digits, and $e$ as describing the location of the “binary point” within this sequence. The use of floating representation is the reason why in many programming systems, printing the expression $0.1+0.2$ will result in $0.30000000000000004$ and not $0.3$, see here, here and here for more.

The reader might be (rightly) worried about the fact that the floating point representation (or the rational number one) can only approximately represent real numbers. In many (though not all) computational applications, one can make the accuracy tight enough so that this does not affect the final result, though sometimes we do need to be careful. Indeed, floating-point bugs can sometimes be no joking matter. A floating point error has been implicated in the explosion of the Ariane 5 rocket, a bug that cost more than 370 million dollars, and the failure of a U.S. Patriot missile to intercept an Iraqi Scud missile, costing 28 lives. Floating point is often problematic in financial applications as well.

2.3.1 Can we represent reals exactly?

Given the issues with floating point representation, we could ask whether we could represent real numbers exactly as strings. Unfortunately, the following theorem shows that this cannot be done:

**Theorem 2.6 — Reals are uncountable.** There is no one-to-one function $RtS : \mathbb{R} \to \{0, 1\}^*$.  

Theorem 2.6 was proven by Georg Cantor in 1874. This result (and the theory around it) was quite shocking to mathematicians at the time. By showing that there is no one-to-one map from $\mathbb{R}$ to $\{0, 1\}^*$ (or $\mathbb{N}$), Cantor showed that these two infinite sets have “different forms of infinity” and that the set of real numbers $\mathbb{R}$ is in some sense “bigger” than the infinite set $\{0, 1\}^*$. The notion that there are “shades of infinity” was deeply disturbing to mathematicians and philosophers at the time. The philosopher Ludwig Wittgenstein (whom we mentioned before) called Cantor’s results “utter nonsense” and “laughable.” Others thought they were even worse than that. Leopold Kronecker called Cantor a “corrupter of youth,” while Henri Poincaré said that Cantor’s ideas “should be banished from mathematics once and for all.” The tide eventually turned, and these days Cantor’s work
is universally accepted as the cornerstone of set theory and the foundations of mathematics. As David Hilbert said in 1925, “No one shall expel us from the paradise which Cantor has created for us.” As we will see later in this book, Cantor’s ideas also play a huge role in the theory of computation.

Now that we have discussed the theorem’s importance, let us see the proof. The idea behind the proof is to do the following:

1. Define some infinite set \( \mathcal{X} \) for which it is easier for us to prove that \( \mathcal{X} \) is not countable (namely, it’s easier for us to prove is there is no one-to-one function from \( \mathcal{X} \) to \( \{0, 1\}^* \)).

2. Prove that there is a one-to-one function \( G \) mapping \( \mathcal{X} \) to \( \mathbb{R} \).

These two facts together imply Cantor’s Theorem. Indeed, we can show this implication using a “proof by contradiction.” If we assume (towards the sake of contradiction) that there exists some one-to-one \( F \) mapping \( \mathbb{R} \) to \( \{0, 1\}^* \) then the function \( x \mapsto F(G(x)) \) obtained by composing \( F \) with the function \( G \) from Step 2 above would be a one-to-one function from \( \mathcal{X} \) to \( \{0, 1\}^* \), which contradicts what we proved in Step 1!

To turn this idea into a proof of Theorem 2.6 we need to:

• Define the set \( \mathcal{X} \).

• Prove that there is no one-to-one function from \( \mathcal{X} \) to \( \{0, 1\}^* \).

• Prove that there is a one-to-one function from \( \mathcal{X} \) to \( \mathbb{R} \).

We now proceed to do precisely that. That is, we will present a definition for a certain set and then state and prove two lemmas that show that this set satisfies our two desired properties.

**Definition 2.7** — **The set** \( \{0, 1\}^\infty \). We denote by \( \{0, 1\}^\infty \) the set \( \{ f \mid f : \mathbb{N} \to \{0, 1\} \} \).

That is, \( \{0, 1\}^\infty \) is a set of **functions**, and a function \( f \) is in \( \{0, 1\}^\infty \) if and only if its domain is \( \mathbb{N} \) and its codomain is \( \{0, 1\} \). The set \( \{0, 1\}^\infty \) will play the role of \( \mathcal{X} \) above. Namely, we will prove the following two results about it:

**Lemma 2.8** There does not exist a one-to-one map \( F_{tS} : \{0, 1\}^\infty \to \{0, 1\}^* \).\(^{16}\)

**Lemma 2.9** There does exist a one-to-one map \( F_{tR} : \{0, 1\}^\infty \to \mathbb{R} \).\(^ {17}\)

As we’ve seen above, **Lemma 2.8** and **Lemma 2.9** together imply **Theorem 2.6**. To repeat the argument more formally, suppose, for the sake of contradiction, that there did exist a one-to-one function

\(^{15}\) We can also think of \( \{0, 1\}^\infty \) as the set of all infinite sequences of bits, since a function \( f : \mathbb{N} \to \{0, 1\} \) can be identified with the sequence \( f(0), f(1), f(2), \ldots \).

\(^{16}\) \( F_{tS} \) stands for “functions to strings.”

\(^{17}\) \( F_{tR} \) stands for “functions to reals.”
Computation and Representation

Figure 2.7: We prove Theorem 2.6 by combining Lemma 2.8 and Lemma 2.9. Lemma 2.9, which uses standard calculus tools, shows the existence of a one-to-one map \( F_t \) from the set of finite sequences of \( \{0, 1\} \) to the real numbers. So, if a hypothetical one-to-one map \( R_t \) existed, then we could compose them to get a one-to-one map \( F_t \). Yet this contradicts Lemma 2.8—the heart of the proof—which rules out the existence of such a map.

Proof. We will prove that there does not exist an onto function \( StF : \{0, 1\} \to \{0, 1\} \). This will imply the lemma since for every two sets \( A \) and \( B \), there exists an onto function from \( A \) to \( B \) if and only if there exists a one-to-one function from \( B \) to \( A \) (see Lemma 1.6).

Let \( StF : \{0, 1\} \to \{0, 1\} \) be any function mapping \( \{0, 1\} \) to \( \{0, 1\} \). We will prove that \( StF \) is not onto by showing that there exists some \( f^* \in \{0, 1\} \) that is not in the image of the function \( StF \). Namely, \( StF(x) \neq f^* \) for every \( x \in \{0, 1\} \).

The construction of \( f^* \) is short but subtle. For every number \( n \in \mathbb{N} \), we let \( x(n) \) be the string obtained by representing \( n \) in the binary basis and “chopping off” its most significant digit. We define \( f^*(n) \) as follows:

\[
f^*(n) = 1 - StF(x(n))(n) \tag{2.4}
\]

As computer scientists, let’s first verify that (2.4) “type checks”. First of all, \( f^* \) is a member of \( \{0, 1\} \) and so for every \( n \in \mathbb{N} \), \( f^*(n) \) should be a bit in \( \{0, 1\} \), and so for (2.4) to “type check” we need its right-hand side to also be a bit. For every \( n \), \( x(n) \) is a string in \( \{0, 1\} \) and so \( StF(x(n)) \) is a function \( g \in \{0, 1\} \) of any. If we apply the function \( g = StF(x(n)) \) to \( n \) we get a bit \( b \in \{0, 1\} \) and so \( 1 - b \) is indeed also a bit as we needed it to be.

Now we want to prove that for every \( x \in \{0, 1\} \), \( StF(x) \neq f^* \). Indeed, suppose (towards a contradiction) that there did exist some \( x \in \{0, 1\} \) such that

\[
StF(x) = f^* . \tag{2.5}
\]

Then, if we let \( n \) be the number whose binary representation is \( 1x \), we see that one the one hand by (2.5) \( f^*(n) = StF(x)(n) \) but on the other hand (since \( x(n) = x \)) by (2.4)

\[
f^*(n) = 1 - StF(x)(n) . \tag{2.6}
\]

We obtained that \( f^*(n) \) is equal to both \( StF(x)(n) \) and to one minus the same quantity which is clearly a contradiction!
The proof of Lemma 2.9 is rather subtle, and worth re-reading a second or third time. It is known as the “diagonal” argument, as the construction of \( f^* \) can be thought of as going over the diagonal elements of a table that in the \( n \)-th row and \( m \)-column contains \( StF(x)(m) \) where \( x \) is the string such that \( n(x) = n \), see Fig. 2.8. We will use the diagonal argument again several times later on in this book.

Remark 2.10 — Generalizing beyond strings and reals.

Lemma 2.8 doesn’t really have much to do with the natural numbers or the strings. An examination of the proof shows that it really shows that for every set \( S \), there is no one-to-one map \( F : \{0, 1\}^S \rightarrow S \) where \( \{0, 1\}^S \) denotes the set \( \{ f \mid f : S \rightarrow \{0, 1\} \} \) of all Boolean functions with domain \( S \). Since we can identify a subset \( V \subseteq S \) with its characteristic function \( f = 1_V \) (i.e., \( 1_V(x) = 1 \) iff \( x \in V \)), we can think of \( \{0, 1\}^S \) also as the set of all subsets of \( S \). This subset is sometimes called the power set of \( S \).

The proof of Lemma 2.8 can be generalized to show that there is no one-to-one map between a set and its power set. In particular, it means that the set \( \{0, 1\}^\omega \) is “even bigger” than \( \mathbb{N} \). Cantor used these ideas to construct an infinite hierarchy of shades of infinity. The number of such shades turns out to be much larger than \( |\mathbb{N}| \) or even \( |\mathbb{R}| \). He denoted the cardinality of \( \mathbb{N} \) by \( \aleph_0 \) and denoted the next largest infinite number by \( \aleph_1 \). (\( \aleph \) is the first letter in the Hebrew alphabet.) Cantor also made the continuum hypothesis that \( |\mathbb{R}| = \aleph_1 \). We will come back to the fascinating story of this hypothesis later on in this book. This lecture of Aaronson mentions some of these issues (see also this Berkeley CS 70 lecture).

To complete the proof of Theorem 2.6, we need to show Lemma 2.9. This requires some calculus background but is otherwise straightforward. The idea is that we can construct a one-to-one map from \( \{0, 1\}^\omega \) to the real numbers by mapping the function \( f : \mathbb{N} \rightarrow \{0, 1\} \) to the number that has the infinite decimal expansion \( f(0).f(1)f(2)f(3)f(4)f(5)\ldots \) (i.e., the number between 0 and 2 that is \( \sum_{i=0}^{\infty} f(i)10^{-i} \)). We will now do this more formally. If you have not had much experience with limits of a real series before, then the formal proof below might be a little hard to follow. This part is not the core of Cantor’s argument, nor are such limits important to the remainder of this book, so feel free to also just take Lemma 2.9 on faith and skip the proof.
You could wonder why we can’t immediately deduce that two numbers that differ in a digit are not the same. The issue is that we have to be a little more careful when talking about infinite expansions. For example, the number half has two decimal expansions \(0.5\) and \(0.49999\ldots\). However, this issue does not come up if (as in our case) we restrict attention only to numbers with decimal expansions that do not involve the digit \(9\).

Proof Idea:

As discussed above, we define \(\text{FtR}(f)\) to be the number between \(0\) and \(2\) whose decimal expansion is \(f(0).f(1)f(2)\ldots\), or in other words \(\text{FtR}(f) = \sum_{i=0}^{\infty} f(i) \cdot 10^{-i}\). To prove that \(\text{FtR}\) is one to one, we need to show that if \(f \neq g\) then \(\text{FtR}(f) \neq \text{FtR}(g)\). To do that we let \(k \in \mathbb{N}\) be the first input on which \(f\) and \(g\) disagree. Then the numbers \(\text{FtR}(f)\) and \(\text{FtR}(g)\) agree in the first \(k-2\) digits following the decimal point and disagree in the \(k-1\)-th digit. One can then calculate and verify that this means that \(|\text{FtR}(f) - \text{FtR}(g)| > 0.5 \cdot 10^{-k}\) which in particular means that these two numbers are distinct from one another.\(^{18}\)

* Proof of Lemma 2.9. For every \(f \in \{0, 1\}^\infty\) and \(n \in \mathbb{N}\), we define \(S(f)_n = \sum_{i=0}^{n} f(i)10^{-i}\). It is a known result in calculus (whose proof we will not repeat here) that for every \(f : \mathbb{N} \to \{0, 1\}\), the sequence \((S(f)_n)_{n=0}^{\infty}\) has a limit. In other words, for every \(f\) there is a value \(\alpha(f) \in \mathbb{R}\) such that for every \(\epsilon > 0\), if \(n\) is sufficiently large then \(|S(f)_n - \alpha| < \epsilon\). The value \(\alpha(f)\) is denoted by \(\sum_{i=0}^{\infty} f(i) \cdot 10^{-i}\). We define the function \(\text{FtR}\) by setting \(\text{FtR}(f) = \alpha(f)\).

We define \(\text{FtR}(f)\) to be this value \(x(f)\). In other words, we define

\[
\text{FtR}(f) = \sum_{i=0}^{\infty} f(i) \cdot 10^{-i}
\]  

which will be a number between \(0\) and \(2\).

To show that \(\text{FtR}\) is one to one, we need to show that \(\text{FtR}(f) \neq \text{FtR}(g)\) for every distinct \(f, g : \mathbb{N} \to \{0, 1\}\). Let \(f \neq g\) be such functions. Since \(f\) and \(g\) are distinct, there must be some input on which they differ, and we define \(k\) to be the smallest such input. That is, \(k \in \mathbb{N}\) is the smallest number for which \(f(k) \neq g(k)\). We will show that \(|\text{FtR}(f) - \text{FtR}(g)| > 0.5 \cdot 10^{-k}\). This will complete the proof since in particular it implies that \(\text{FtR}(f) \neq \text{FtR}(g)\).

Since \(f(k) \neq g(k)\), we can assume without loss of generality that \(f(k) = 0\) and \(g(k) = 1\) (otherwise, if \(f(k) = 1\) and \(g(k) = 1\), then we can simply switch the roles of \(f\) and \(g\)). Define \(S = \sum_{i=0}^{k-1} 10^{-i} \cdot f(i) = \sum_{i=0}^{k-1} 10^{-i} \cdot g(i)\) (the equality holds since \(f\) and \(g\) agree up to \(k-1\)). Now, since \(g(k) = 1\), we can write

\[
\text{FtR}(g) = \sum_{i=0}^{\infty} g(i)10^{-i} \geq S + g(k)10^{-k} = S + 10^{-k}.
\]  

On the other hand, since \(f(k) = 0\),

\[
\text{FtR}(f) = \sum_{i=0}^{\infty} f(i)10^{-i} = S + \sum_{i=k+1}^{\infty} f(i)10^{-i} \leq S + 10^{-(k-1)} \sum_{j=0}^{\infty} 10^{-j}.
\]  

\(^{18}\) You could wonder why we can’t immediately deduce that two numbers that differ in a digit are not the same. The issue is that we have to be a little more careful when talking about infinite expansions. For example, the number half has two decimal expansions \(0.5\) and \(0.49999\ldots\). However, this issue does not come up if (as in our case) we restrict attention only to numbers with decimal expansions that do not involve the digit \(9\).
Now \( \sum_{j=0}^{\infty} 10^{-j} \) is simply the number 1.11111 ... = 11/9, and hence we get that

\[
F_{tr}(f) \leq S + 11/9 \cdot 10^{-k-1} = S + 11/90 < S + 0.2 \cdot 10^{-k} \quad (2.10)
\]

while \( F_{tr}(g) \geq S + 10^{-k} \) which means the difference between them is larger than \( 0.5 \cdot 10^{-k} \).

### 2.4 BEYOND NUMBERS

We can of course represent objects other than numbers as binary strings. Let us give a general definition for a representation scheme. Such a scheme for representing objects from some set \( \mathcal{O} \) consists of an encoding function that maps an object in \( \mathcal{O} \) to a string, and a decoding function that decodes a string back to an object in \( \mathcal{O} \). Formally, we make the following definition:

**Definition 2.11** — **String representation.** Let \( \mathcal{O} \) be some set. A representation scheme for \( \mathcal{O} \) is a pair of functions \( E, D \) where \( E : \mathcal{O} \rightarrow \{0, 1\}^* \) is a total one-to-one function, \( D : \{0, 1\}^* \rightarrow_p \mathcal{O} \) is a (possibly partial) function, and such that \( D(E(o)) = o \) for every \( o \in \mathcal{O} \). \( E \) is known as the encoding function and \( D \) is known as the decoding function.

Note that the condition \( D(E(o)) = o \) for every \( o \in \mathcal{O} \) implies that \( D \) is onto (can you see why?). It turns out that to construct a representation scheme we only need to find an encoding function. That is, every one-to-one encoding function has a corresponding decoding function, as shown in the following lemma:

**Lemma 2.12** Suppose that \( E : \mathcal{O} \rightarrow \{0, 1\}^* \) is one-to-one. Then there exists a function \( D : \{0, 1\}^* \rightarrow \mathcal{O} \) such that \( D(E(o)) = o \) for every \( o \in \mathcal{O} \).

**Proof.** Let \( o_0 \) be some arbitrary element of \( \mathcal{O} \). For every \( x \in \{0, 1\}^* \), there exists either zero or a single \( o \in \mathcal{O} \) such that \( E(o) = x \) (otherwise \( E \) would not be one-to-one). We will define \( D(x) \) to equal \( o_0 \) in the first case and this single object \( o \) in the second case. By definition \( D(E(o)) = o \) for every \( o \in \mathcal{O} \).

**Remark 2.13** — While the decoding function of a representation scheme can in general be a partial function, the proof of **Lemma 2.12** implies that every repre-
2.4.1 Finite representations

If \( \mathcal{O} \) is finite, then we can represent every object in \( o \) as a string of length at most some number \( n \). What is the value of \( n \)? Let us denote by \( \{0, 1\}^n \) the set \( \{x \in \{0, 1\}^* : |x| \leq n\} \) of strings of length at most \( n \). The size of \( \{0, 1\}^n \) is equal to

\[
|\{0, 1\}^0| + |\{0, 1\}^1| + |\{0, 1\}^2| + \cdots + |\{0, 1\}^n| = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1. \tag{2.11}
\]

using the standard formula for summing a geometric progression.

To obtain a representation of objects in \( \mathcal{O} \) as strings of length at most \( n \) we need to come up with a one-to-one function from \( \mathcal{O} \) to \( \{0, 1\}^n \). We can do so, if and only if \( |\mathcal{O}| \leq 2^{n+1} - 1 \) as is implied by the following lemma:

Lemma 2.14 For every two finite sets \( S, T \), there exists a one-to-one \( E : S \rightarrow T \) if and only if \( |S| \leq |T| \).

Proof. Let \( k = |S| \) and \( m = |T| \) and so write the elements of \( S \) and \( T \) as \( S = \{s_0, s_1, \ldots, s_{k-1}\} \) and \( T = \{t_0, t_1, \ldots, t_{m-1}\} \). We need to show that there is a one-to-one function \( E : S \rightarrow T \) if and only if \( k \leq m \). For the “if” direction, if \( k \leq m \) we can simply define \( E(s_i) = t_i \) for every \( i \in [k] \). Clearly for \( i \neq j \), \( t_i = E(s_i) \neq E(s_j) = t_j \), and hence this function is one-to-one. In the other direction, suppose that \( k > m \) and \( E : S \rightarrow T \) is some function. Then \( E \) cannot be one-to-one. Indeed, for \( i = 0, 1, \ldots, m-1 \) let us “mark” the element \( t_j = E(s_i) \) in \( T \). If \( t_j \) was marked before, then we have found two objects in \( S \) mapping to the same element \( t_j \). Otherwise, since \( T \) has \( m \) elements, when we get to \( i = m-1 \) we mark all the objects in \( T \). Hence, in this case, \( E(s_m) \) must map to an element that was already marked before.\(^{19} \)

\[ \]

2.4.2 Prefix-free encoding

When showing a representation scheme for rational numbers, we used the “hack” of encoding the alphabet \( \{0, 1, \| \} \) to represent tuples of strings as a single string. This turns out to be a special case of the general paradigm of prefix-free encoding. The idea is the following: if our representation has the property that no string \( x \) representing an object \( o \) is a prefix (i.e., an initial substring) of a string \( y \) representing a different object \( o' \), then we can represent a lists of objects by merely concatenating the representations of all the list members. For example, because in English every sentence ends with a punctuation mark such
as a period, exclamation, or question mark, we can represent a list of sentences (i.e., a paragraph) by merely concatenating the sentences one after the other.

It turns out that we can transform every representation to a prefix-free form. This justifies Big Idea 1, and allows us to transform a representation scheme for objects of a type \( T \) to a representation scheme of lists of objects of the type \( T \). By repeating the same technique, we can also represent lists of lists of objects of type \( T \), and so on and so forth. But first let us formally define prefix-freeness:

**Definition 2.15 — Prefix free encoding.** For two strings \( y, y' \), we say that \( y \) is a prefix of \( y' \) if \( |y| \leq |y'| \) and for every \( i < |y'|, y'_i = y_i \).

Let \( E : \mathcal{O} \rightarrow \{0,1\}^* \) be a function. We say that \( E \) prefix-free if there does not exist a distinct pair of objects \( o, o' \in \mathcal{O} \) such that \( E(o) \) is a prefix of \( E(o') \).

Recall that for every set \( \mathcal{O} \), the set \( \mathcal{O}^* \) consists of all finite length tuples (i.e., lists) of elements in \( \mathcal{O} \). The following theorem shows that if \( E \) is a prefix-free encoding of \( \mathcal{O} \) then by concatenating encodings we can obtain a valid (i.e., one-to-one) representation of \( \mathcal{O}^* \):

**Theorem 2.16 — Prefix-free implies tuple encoding.** Suppose that \( E : \mathcal{O} \rightarrow \{0,1\}^* \) is prefix-free. Then the following map \( E : \mathcal{O}^* \rightarrow \{0,1\}^* \) is one to one, for every \( o_0, \ldots, o_{k-1} \in \mathcal{O}^* \), we define

\[
E(o_0, \ldots, o_{k-1}) = E(o_0)E(o_1) \cdots E(o_{k-1}).
\]

(2.12)

**Proof Idea:**

The idea behind the proof is simple. Suppose that for example we want to decode a triple \( (o_0, o_1, o_2) \) from its representation \( x = E'(o_0, o_1, o_2) = E(o_0)E(o_1)E(o_2) \). We will do so by first finding the first prefix \( x_0 \) of \( x \) such is a representation of some object. Then we will decode this object, remove \( x_0 \) from \( x \) to obtain a new string \( x' \), and continue onwards to find the first prefix \( x_1 \) of \( x' \) and so on and so forth (see Exercise 2.8). The prefix-freeness property of \( E \) will ensure that \( x_0 \) will in fact be \( E(o_0) \), \( x_1 \) will be \( E(o_1) \) etc.
Proof of Theorem 2.16. We now show the formal proof. Suppose, towards the sake of contradiction that there exist two distinct tuples $(o_0, \ldots, o_{k-1})$ and $(o'_0, \ldots, o'_{k'-1})$ such that

$$E(o_0, \ldots, o_{k-1}) = E(o'_0, \ldots, o'_{k'-1}).$$

(2.13)

We will denote the string $E(o_0, \ldots, o_{k-1})$ by $\overline{x}$.

Let $i$ be the first coordinate such that $o_i \neq o'_i$. (If $o_i = o'_i$ for all $i$ then, since we assume the two tuples are distinct, one of them must be larger than the other. In this case we assume without loss of generality that $k' > k$ and let $i = k$.) In the case that $i < k$, we see that the string $\overline{x}$ can be written in two different ways:

$$\overline{x} = E(o_0, \ldots, o_{k-1}) = x_0 \cdots x_{i-1} E(o_i) E(o_{i+1}) \cdots E(o_{k-1})$$

(2.14)

and

$$\overline{x} = E(o'_0, \ldots, o'_{k'-1}) = x_0 \cdots x_{i-1} E(o'_i) E(o'_{i+1}) \cdots E(o'_{k'-1})$$

(2.15)

where $x_j = E(o_j) = E(o'_j)$ for all $j < i$. Let $\overline{y}$ be the string obtained after removing the prefix $x_0 \cdots x_{i-1}$ from $\overline{x}$. We see that $\overline{y}$ can be written as both $\overline{y} = E(o_i)s$ for some string $s \in \{0, 1\}^*$ and as $\overline{y} = E(o'_i)s'$ for some $s' \in \{0, 1\}^*$. But this means that one of $E(o_i)$ and $E(o'_i)$ must be a prefix of the other, contradicting the prefix-freeness of $E$.

In the case that $i = k$ and $k' > k$, we get a contradiction in the following way. In this case

$$\overline{x} = E(o_0) \cdots E(o_{k-1}) = E(o_0) \cdots E(o_{k-1}) E(o'_k) \cdots E(o'_{k'-1})$$

(2.16)

which means that $E(o'_k) \cdots E(o'_{k'-1})$ must correspond to the empty string "". But in such a case $E(o'_k)$ must be the empty string, which in particular is the prefix of any other string, contradicting the prefix-freeness of $E$.

\[ \square \]

Remark 2.17 — Prefix freeness of list representation.

Even if the representation $E$ of objects in $\mathcal{O}$ is prefix free, this does not mean that our representation $\overline{E}$ of lists of such objects will be prefix free as well. In fact, it won’t be: for every three objects $o, o', o''$ the representation of the list $(o, o')$ will be a prefix of the representation of the list $(o, o', o'')$. However, as we see
in Lemma 2.18 below, we can transform every representation into prefix-free form, and so will be able to use that transformation if needed to represent lists of lists, lists of lists of lists, and so on and so forth.

2.4.3 Making representations prefix-free

Some natural representations are prefix-free. For example, every fixed output length representation (i.e., one-to-one function \( E : \mathcal{O} \to \{0, 1\}^n \)) is automatically prefix-free, since a string \( x \) can only be a prefix of an equal-length \( x' \) if \( x \) and \( x' \) are identical. Moreover, the approach we used for representing rational numbers can be used to show the following:

**Lemma 2.18** Let \( E : \mathcal{O} \to \{0, 1\}^* \) be a one-to-one function. Then there is a one-to-one prefix-free encoding \( \overline{E} : \mathcal{O} \to \{0, 1\}^* \) such that \( |\overline{E}(o)| \leq 2|E(o)| + 2 \) for every \( o \in \mathcal{O} \).

For the sake of completeness, we will include the proof below, but it is a good idea for you to pause here and try to prove it on your own, using the same technique we used for representing rational numbers.

**Proof of Lemma 2.18.** Define the function \( PF : \{0, 1\}^* \to \{0, 1\}^* \) as follows \( PF(x) = x_0x_0x_1x_1 \ldots x_{n-1}x_{n-1}.01 \) for every \( x \in \{0, 1\}^* \). If \( E : \mathcal{O} \to \{0, 1\}^* \) is the (potentially not prefix-free) representation for \( \mathcal{O} \), then we transform it into a prefix-free representation \( \overline{E} : \mathcal{O} \to \{0, 1\}^* \) by defining \( \overline{E}(o) = PF(E(o)) \).

To prove the lemma we need to show that (1) \( \overline{E} \) is one-to-one and (2) \( \overline{E} \) is prefix-free. In fact (2) implies (1), since if \( \overline{E}(o) \) is never a prefix of \( \overline{E}(o') \) for every \( o \neq o' \) then in particular \( \overline{E} \) is one-to-one. Now suppose, toward a contradiction, that there are \( o \neq o' \) in \( \mathcal{O} \) such that \( \overline{E}(o) \) is a prefix of \( \overline{E}(o') \). (That is, if \( y = \overline{E}(o) \) and \( y' = \overline{E}(o') \), then \( y_j = y'_j \) for every \( j < |y| \).)

Define \( x = E(o) \) and \( x' = E(o') \). Note that since \( E \) is one-to-one, \( x \neq x' \). (Recall that two strings \( x, x' \) are distinct if they either differ in length or have at least one distinct coordinate.) Under our assumption, \( |PF(x)| \leq |PF(x')| \), and since by construction \( |PF(x)| = 2|x| + 2 \), it follows that \( |x| \leq |x'| \). If \( |x| = |x'| \) then, since \( x \neq x' \), there must be a coordinate \( i \in \{0, \ldots, |x| - 1\} \) such that \( x_i \neq x'_i \). But since \( PF(x)_{2i} = x_i \), we get that \( PF(x)_{2i} \neq PF(x')_{2i} \), and hence \( E(o) = PF(x) \) is not a prefix of \( E(o') = PF(x') \). Otherwise (if \( |x| \neq |x'| \)) then it must be that \( |x| < |x'| \), and hence if \( n = |x| \), then \( PF(x)_{2n} = 0 \) and \( PF(x)_{2n+1} = 1 \). But since \( n < |x'| \), \( PF(x')_{2n}, PF(x')_{2n+1} \) is equal to
either 00 or 11, and in any case we get that \( E(o) = PF(x) \) is not a prefix of \( E(o') = PF(x') \).

The proof of Lemma 2.18 is not the only or even the best way to transform an arbitrary representation into prefix-free form. In fact, we can even obtain a more efficient transformation satisfying \( |E'(o)| \leq |o| + O(\log |o|) \). We leave proving this as an exercise (see Exercise 2.9).

2.4.4 “Proof by Python” (optional)
The proofs of Theorem 2.16 and Lemma 2.18 are constructive in the sense that they give us:

- A way to transform the encoding and decoding functions of any representation of an object \( O \) to an encoding and decoding functions that are prefix-free;

- A way to extend prefix-free encoding and decoding of single objects to encoding and decoding of lists of objects by concatenation.

Specifically, we could transform any pair of Python functions encode and decode to functions pfencode and pfdecode that correspond to a prefix-free encoding and decoding. Similarly, given pfencode and pfdecode for single objects, we can extend them to encoding of lists. Let us show how this works for the case of the \( \text{NtS} \) and \( \text{StN} \) functions we defined above.

We start with the “Python proof” of Lemma 2.18: a way to transform an arbitrary representation into one that is prefix free. The function \texttt{prefixfree} below takes as input a pair of encoding and decoding functions, and returns a triple of functions containing prefix-free encoding and decoding functions, as well as a function that checks whether a string is a valid encoding of an object.

```python
# takes functions encode and decode mapping
# objects to lists of bits and vice versa,
# and returns functions pfencode and pfdecode that
# maps objects to lists of bits and vice versa
# in a prefix-free way.
# Also returns a function pfvalid that says
# whether a list is a valid encoding

def prefixfree(encode, decode):
    def pfencode(o):
        L = encode(o)
        return [L[i//2] for i in range(2*len(L))]+[0,1]
    def pfdecode(L):
        return decode([L[j] for j in range(0,len(L)-2,2)])
```

When it’s not too awkward, we use the term “Python function” or “subroutine” to distinguish between such snippets of python programs and mathematical functions. However, in comments in python source we use “functions” to denote python functions, just as we use “integers” to denote python int objects.

```python
def pfvalid(L):
    return (len(L) % 2 == 0) and L[-2:]==[0,1]
return pfencode, pfdecode, pfvalid
pfNtS, pfStN, pfvalidN = prefixfree(NtS,StN)
```

```
NtS(234)
  # 11101010
pfNtS(234)
  # 11111010110110110
pfStN(pfNtS(234))
  # 234
pfvalidM(pfNtS(234))
  # true
```

Note that Python function prefixfree above takes two Python functions as input and outputs three Python functions as output. You don’t have to know Python in this course, but you do need to get comfortable with the idea of functions as mathematical objects in their own right, that can be used as inputs and outputs of other functions.

We now show a “Python proof” of Theorem 2.16. Namely, we show a function represlists that takes as input a prefix-free representation scheme (implemented via encoding, decoding, and validity testing functions) and outputs a representation scheme for lists of such objects. If we want to make this representation prefix-free then we could fit it into the function prefixfree above.

```
# Takes functions pfencode, pfdecode and pfvalid,
# and returns functions encodelists, decodelist
# that can encode and decode
# lists of the objects respectively
def represlists(pfencode,pfdecode,pfvalid):
    def encodelist(L):
        """Gets list of objects, encodes it as list of bits""
        return ''.join([pfencode(obj) for obj in L])
    def decodelist(S):
        """Gets lists of bits, returns lists of objects""
```

20 When it’s not too awkward, we use the term “Python function” or “subroutine” to distinguish between such snippets of python programs and mathematical functions. However, in comments in python source we use “functions” to denote python functions, just as we use “integers” to denote python int objects.
\[ i=0; j=1 \text{; res = []} \]
\[ \textbf{while} \ j<\text{len}(S): \]
\[ \text{if} \ \text{pfvalid}(S[i:j]): \]
\[ \text{res} \text{+= \{pfdecode(S[i:j])\}} \]
\[ i=j \]
\[ j+=1 \]
\[ \textbf{return} \ \text{res} \]

\[ \text{return} \ \text{enclist, decodelist} \]

\[ \texttt{LtS, StL = represlists(pfNtS, pfStN, pfvalidN)} \]

\[ \texttt{LtS([234, 12, 5])} \]
\[ \# \ 111111001100110001111100000111001101 \]
\[ \texttt{StL(LtS([234, 12, 5]))} \]
\[ \# \ [234, 12, 5] \]

### 2.4.5 Representing letters and text

We can represent a letter or symbol by a string, and then if this representation is prefix-free, we can represent a sequence of symbols by merely concatenating the representation of each symbol. One such representation is the ASCII that represents 128 letters and symbols as strings of 7 bits. Since the ASCII representation is fixed-length, it is automatically prefix-free (can you see why?). Unicode is representation of (at the time of this writing) about 128,000 symbols as numbers (known as code points) between 0 and 1,114,111. There are several types of prefix-free representations of the code points, a popular one being UTF-8 that encodes every codepoint into a string of length between 8 and 32.

- **Example 2.19 — The Braille representation.** The Braille system is another way to encode letters and other symbols as binary strings. Specifically, in Braille, every letter is encoded as a string in \{0, 1\}^6, which is written using indented dots arranged in two columns and three rows, see Fig. 2.10. (Some symbols require more than one six-bit string to encode, and so Braille uses a more general prefix-free encoding.)

  The Braille system was invented in 1821 by Louis Braille when he was just 12 years old (though he continued working on it and improving it throughout his life). Braille was a French boy that lost his eyesight at the age of 5 as the result of an accident.
Example 2.20 — Representing objects in C (optional). We can use programming languages to probe how our computing environment represents various values. This is easiest to do in “unsafe” programming languages such as C that allow direct access to the memory.

Using a simple C program we have produced the following representations of various values. One can see that for integers, multiplying by 2 corresponds to a “left shift” inside each byte. In contrast, for floating point numbers, multiplying by two corresponds to adding one to the exponent part of the representation. A negative number is represented using the two’s complement approach. Strings are represented in a prefix-free form by ensuring that a zero byte is at their end.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *bytes(void *p, int n){
    int i;
    int j;
    char *a = (char *) p;
    char *s = malloc(9*n+2);
    for (i = 0; i < n; ++i){
        s[i*9] = a[i];
        s[i*9+1] = a[i];
        s[i*9+2] = a[i];
        s[i*9+3] = a[i];
        s[i*9+4] = a[i];
        s[i*9+5] = a[i];
        s[i*9+6] = a[i];
        s[i*9+7] = a[i];
        s[i*9+8] = a[i];
    }
    return s;
}
```

If you are curious, the code for this program (which you can run here) is the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *bytes(void *p, int n){
    int i;
    int j;
    char *a = (char *) p;
    char *s = malloc(9*n+2);
    for (i = 0; i < n; ++i){
        s[i*9] = a[i];
        s[i*9+1] = a[i];
        s[i*9+2] = a[i];
        s[i*9+3] = a[i];
        s[i*9+4] = a[i];
        s[i*9+5] = a[i];
        s[i*9+6] = a[i];
        s[i*9+7] = a[i];
        s[i*9+8] = a[i];
    }
    return s;
}
```
```c
s[9*n] = '
';
s[9*n+1] = 0;

j = 0;
for(i=0; i< n*8; i++){
    s[j] = a[i/8] & (128 >> (i % 8)) ? '1' : '0';
    if (i% 8 == 7) { s[++j] = ' '; }
    ++j;
    return s;
}

void printint(int a) {
    printf("%-8s %-5d: %s", "int", a,
            bytes(&a,sizeof(int)));
}

void printlong(long a) {
    printf("%-8s %-5d: %s", "long", a,
            bytes(&a,sizeof(long)));
}

void printstring(char *s) {
    printf("%-8s %-5s: %s", "string", s,
            bytes(s,strlen(s)+1));
}

void printfloat(float f) {
    printf("%-8s %-5.1f: %s", "float", f,
            bytes(&f,sizeof(float)));
}

void printdouble(double f) {
    printf("%-8s %-5.1f: %s", "double", f,
            bytes(&f,sizeof(double)));
}

int main(void) {
```
2.4.6 Representing vectors, matrices, images

Once we can represent numbers and lists of numbers, then we can also represent vectors (which are just lists of numbers). Similarly, we can represent lists of lists, and thus, in particular, can represent matrices. To represent an image, we can represent the color at each pixel by a list of three numbers corresponding to the intensity of Red, Green and Blue. Thus an image of \( n \) pixels would be represented by a list of \( n \) such length-three lists. A video can be represented as a list of images.\(^{21}\)

2.4.7 Representing graphs

A graph on \( n \) vertices can be represented as an \( n \times n \) adjacency matrix whose \((i,j)\)th entry is equal to 1 if the edge \((i,j)\) is present and is equal to 0 otherwise. That is, we can represent an \( n \) vertex directed graph \( G = (V,E) \) as a string \( A \in \{0,1\}^n \) such that \( A_{i,j} = 1 \) iff the edge \( \vec{ij} \in E \). We can transform an undirected graph to a directed graph by replacing every edge \( \{i,j\} \) with both edges \( \vec{ij} \) and \( \vec{ji} \).

Another representation for graphs is the adjacency list representation. That is, we identify the vertex set \( V \) of a graph with the set \([n]\) where \( n = |V| \), and represent the graph \( G = (V,E) \) as a list of \( n \)

\(^{21}\) We can restrict to three primary colors since (most) humans only have three types of cones in their retinas. We would have needed 16 primary colors to represent colors visible to the Mantis Shrimp.\(^{22}\) Of course these representations are rather wasteful and much more compact representations are typically used for images and videos, though this will not be our concern in this book.\(^{22}\)
lists, where the $i$-th list consists of the out-neighbors of vertex $i$. The difference between these representations can be significant for some applications, though for us would typically be immaterial.

Once again, we can also define these encoding and decoding functions in python:

```python
from graphviz import Graph

# get n by n matrix (as list of n lists)
# return graph corresponding to it
def matrix2graph(M):
    G = Graph(); n = len(M)
    for i in range(n):
        G.node(str(i))  # add vertex i
        for j in range(n):
            if M[i][j]: G.edge(str(i), str(j))  # if M[i][j] is nonzero then add edge between i and j
    return G

matrix2graph([[0, 1, 0], [0, 0, 1], [1, 0, 0]])
```

![Figure 2.11: Representing the graph $G = \{\{0, 1, 2, 3, 4\}, \{(1, 0), (4, 0), (1, 4), (4, 1), (2, 1), (3, 2), (4, 3)\}\}$ in the adjacency matrix and adjacency list representations.](image)

### 2.4.8 Representing lists

If we have a way of representing objects from a set $\mathcal{O}$ as binary strings, then we can represent lists of these objects by applying a prefix-free transformation. Moreover, we can use a trick similar to the above to handle nested lists. The idea is that if we have some representation $E : \mathcal{O} \rightarrow \{0, 1\}^*$, then we can represent nested lists of items from $\mathcal{O}$ using strings over the five element alphabet $\Sigma = \{0, 1, [, , ]\}$.
For example, if $o_1$ is represented by $0011$, $o_2$ is represented by $10011$, and $o_3$ is represented by $00111$, then we can represent the nested list $(o_1, (o_2, o_3))$ as the string $"[0011, [10011, 00111]]"$ over the alphabet $\Sigma$. By encoding every element of $\Sigma$ itself as a three-bit string, we can transform any representation for objects $\mathcal{O}$ into a representation that enables representing (potentially nested) lists of these objects.

2.4.9 Notation
We will typically identify an object with its representation as a string. For example, if $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is some function that maps strings to strings and $x$ is an integer, we might make statements such as “$F(x) + 1$ is prime” to mean that if we represent $x$ as a string $\overline{x}$ and let $\overline{y} = F(\overline{x})$, then the integer $y$ represented by the string $\overline{y}$ satisfies that $y + 1$ is prime. (You can see how this convention of identifying objects with their representation can save us a lot of cumbersome formalism.) Similarly, if $x, y$ are some objects and $F$ is a function that takes strings as inputs, then by $F(x, y)$ we will mean the result of applying $F$ to the representation of the ordered pair $(x, y)$. We will use the same notation to invoke functions on $k$-tuples of objects for every $k$.

This convention of identifying an object with its representation as a string is one that we humans follow all the time. For example, when people say a statement such as “$17$ is a prime number”, what they really mean is that the integer whose decimal representation is the string “$17$”, is prime.

![Figure 2.12: A computational process](image)
focus on computational tasks. That is, we focus on the specification and not the implementation. Again, at an abstract level, a computational task can specify any relation that the output needs to have with the input. However, for most of this book, we will focus on the simplest and most common task of computing a function. Here are some examples:

- **Given** (a representation) of two integers \(x, y\), compute the product \(x \times y\). Using our representation above, this corresponds to computing a function from \(\{0, 1\}^*\) to \(\{0, 1\}^*\). We have seen that there is more than one way to solve this computational task, and in fact, we still do not know the best algorithm for this problem.

- **Given** (a representation of) an integer \(z\), compute its factorization; i.e., the list of primes \(p_1 \leq \cdots \leq p_k\) such that \(z = p_1 \cdots p_k\). This again corresponds to computing a function from \(\{0, 1\}^*\) to \(\{0, 1\}^*\). The gaps in our knowledge of the complexity of this problem are even longer.

- **Given** (a representation of) a graph \(G\) and two vertices \(s\) and \(t\), compute the length of the shortest path in \(G\) between \(s\) and \(t\), or do the same for the longest path (with no repeated vertices) between \(s\) and \(t\). Both these tasks correspond to computing a function from \(\{0, 1\}^*\) to \(\{0, 1\}^*\), though it turns out that there is a vast difference in their computational difficulty.

- **Given** the code of a Python program, determine whether there is an input that would force it into an infinite loop. This task corresponds to computing a partial function from \(\{0, 1\}^*\) to \(\{0, 1\}\) since not every string corresponds to a syntactically valid Python program. We will see that we do understand the computational status of this problem, but the answer is quite surprising.

- **Given** (a representation of) an image \(I\), decide if \(I\) is a photo of a cat or a dog. This correspond to computing some (partial) function from \(\{0, 1\}^*\) to \(\{0, 1\}\).

**Remark 2.21 — Boolean functions and languages.** An important special case of computational tasks corresponds to computing Boolean functions, whose output is a single bit \(\{0, 1\}\). Computing such functions corresponds to answering a YES/NO question, and hence this task is also known as a decision problem. Given any function \(F : \{0, 1\}^* \rightarrow \{0, 1\}\) and \(x \in \{0, 1\}^*\), the task of computing \(F(x)\) corresponds to the task of deciding whether or not \(x \in L\) where \(L = \{x : F(x) = 1\}\) is known as the language that corresponds to the function.
For every particular function $F$, there can be several possible \textit{algorithms} to compute $F$. We will be interested in questions such as:

- For a given function $F$, can it be the case that \textit{there is no algorithm} to compute $F$?

- If there is an algorithm, what is the best one? Could it be that $F$ is “effectively uncomputable” in the sense that every algorithm for computing $F$ requires a prohibitively large amount of resources?

- If we cannot answer this question, can we show equivalence between different functions $F$ and $F'$ in the sense that either they are both easy (i.e., have fast algorithms) or they are both hard?

- Can a function being hard to compute ever be a \textit{good thing}? Can we use it for applications in areas such as cryptography?

In order to do that, we will need to mathematically define the notion of an \textit{algorithm}, which is what we will do in Chapter 3.

\subsection*{2.5.1 Distinguish functions from programs!}
You should always watch out for potential confusions between \textbf{specifications} and \textbf{implementations} or equivalently between \textbf{mathematical functions} and \textbf{algorithms/programs}. It does not help that programming languages (Python included) use the term “\textit{functions}” to denote (parts of) \textit{programs}. This confusion also stems from thousands of years of mathematical history, where people typically defined functions by means of a way to compute them.

For example, consider the multiplication function on natural numbers. This is the function $\text{MULT} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that maps a pair $(x, y)$ of natural numbers to the number $x \cdot y$. As we mentioned, it can be implemented in more than one way:

```python
def mult1(x, y):
    res = 0
    while y>0:
        res += x
        y -= 1
    return res

def mult2(x, y):
    a = NtS(x)
    b = NtS(y)
```

\textbf{Figure 2.13}: A subset $L \subseteq \{0, 1\}^*$ can be identified with the function $F : \{0, 1\}^* \to \{0, 1\}$ such that $F(x) = 1$ if $x \in L$ and $F(x) = 0$ if $x \notin L$. Functions with a single bit of output are called \textit{Boolean functions}, while subsets of strings are called \textit{languages}. The above shows that the two are essentially the same object, and we can identify the task of deciding membership in $L$ (known as \textit{deciding a language} in the literature) with the task of computing the function $F$. 

\textsuperscript{23} The language terminology is due to historical connections between the theory of computation and formal linguistics as developed by Noam Chomsky.
Figure 2.14: A function is a mapping of inputs to outputs. A program is a set of instructions on how to obtain an output given an input. A program computes a function, but it is not the same as a function, popular programming language terminology notwithstanding.

Both `mult1` and `mult2` produce the same output given the same pair of inputs. (Though `mult1` will take far longer to do so when the numbers become large.) Hence, even though these are two different programs, they compute the same mathematical function. This distinction between a program or algorithm \( A \), and the function \( F \) that \( A \) computes will be absolutely crucial for us in this course (see also Fig. 2.14).

**Big Idea 2** A function is not the same as a program. A program computes a function.

Distinguishing functions from programs (or other ways for computing, including circuits and machines) is a crucial theme for this course. For this reason, this is often a running theme in questions that I (and many other instructors) assign in homeworks and exams (hint, hint..).

```
res = 0
res = [0]*(len(a)+len(b))
for i in range(len(a)):
    for j in range(len(b)):
        res += int(a[len(a)-i])*int(b[len(b)-j])*(10**(i+j))
return res

print(mult1(12,7))
# 84
print(mult2(12,7))
# 84
```

This is NOT a function: | This IS a function:
---|---
function even(x) {
    return 1 - (x % 2);
}
into a set of strings \( R(x) \) (for example, \( x \) might describe a set of equations, in which case \( R(x) \) would correspond to the set of all solutions to \( x \)). We can also identify a relation \( R \) with the set of pairs of strings \((x, y)\) where \( y \in R(x) \). A computational process solves a relation if for every \( x \in \{0, 1\}^* \), it outputs some string \( y \in R(x) \).

Later in this book, we will consider even more general tasks, including interactive tasks, such as finding a good strategy in a game, tasks defined using probabilistic notions, and others. However, for much of this book, we will focus on the task of computing a function, and often even a Boolean function, that has only a single bit of output. It turns out that a great deal of the theory of computation can be studied in the context of this task, and the insights learned are applicable in the more general settings.

2.6 EXERCISES

Exercise 2.1 Which one of these objects can be represented by a binary string?

a. An integer \( x \)

b. An undirected graph \( G \).

c. A directed graph \( H \)

d. All of the above.
Exercise 2.2 — More compact than ASCII representation. The ASCII encoding can be used to encode a string of $n$ English letters as a $7n$ bit binary string, but in this exercise, we ask about finding a more compact representation for strings of English lowercase letters.

1. Prove that there exists a representation scheme $(E, D)$ for strings over the 26-letter alphabet $\{a, b, c, \ldots, z\}$ as binary strings such that for every $n > 0$ and length-$n$ string $x \in \{a, b, \ldots, z\}^n$, the representation $E(x)$ is a binary string of length at most $4.8n + 1000$. In other words, prove that for every $n$, there exists a one-to-one function $E : \{a, b, \ldots, z\}^n \rightarrow \{0, 1\}^{\lfloor 4.8n + 1000 \rfloor}$.

2. Prove that there exists no representation scheme for strings over the alphabet $\{a, b, \ldots, z\}$ as binary strings such that for every length-$n$ string $x \in \{a, b, \ldots, z\}^n$, the representation $E(x)$ is a binary string of length $\lfloor 4.6n + 1000 \rfloor$. In other words, prove that there exists some $n > 0$ such that there is no one-to-one function $E : \{a, b, \ldots, z\}^n \rightarrow \{0, 1\}^{\lfloor 4.6n + 1000 \rfloor}$.

3. Python’s `bz2.compress` function is a mapping from strings to strings, which uses the lossless (and hence one to one) bzip2 algorithm for compression. After converting to lowercase, and truncating spaces and numbers, the text of Tolstoy’s “War and Peace” contains $n = 2,517,262$. Yet, if we run `bz2.compress` on the string of the text of “War and Peace” we get a string of length $k = 6,274,768$ bits, which is only $2.49n$ (and in particular much smaller than $4.6n$). Explain why this does not contradict your answer to the previous question.

4. Interestingly, if we try to apply `bz2.compress` on a random string, we get much worse performance. In my experiments, I got a ratio of about 4.78 between the number of bits in the output and the number of characters in the input. However, one could imagine that one could do better and that there exists a company called “Pied Piper” with an algorithm that can losslessly compress a string of $n$ random lowercase letters to fewer than $4.6n$ bits.24 Show that this is not the case by proving that for every $n > 100$ and one to one function $Encode : \{a, \ldots, z\}^n \rightarrow \{0, 1\}^*$, if we let $Z$ be the random variable $|Encode(x)|$ (i.e., the length of $Encode(x)$) for $x$ chosen uniformly at random from the set $\{a, \ldots, z\}^n$, then the expected value of $Z$ is at least $4.6n$.

Exercise 2.3 — Representing graphs: upper bound. Show that there is a string representation of directed graphs with vertex set $[n]$ and degree at most 10 that uses at most $1000n \log n$ bits. More formally, show the

---

24 Actually that particular fictional company uses a metric that focuses more on compression speed than ratio, see here and here.
following. Suppose we define for every \( n \in \mathbb{N} \), the set \( G_n \) as the set containing all directed graphs (with no self loops) over the vertex set \([n]\) where every vertex has degree at most 10. Then, prove that for every sufficiently large \( n \), there exists a one-to-one function \( E : G_n \to \{0, 1\}^{\lceil 1000n \log n \rceil} \).

**Exercise 2.4 — Representing graphs: lower bound.** 1. Define \( S_n \) to be the set of one-to-one and onto functions mapping \([n]\) to \([n]\). Prove that there is a one-to-one mapping from \( S_n \) to \( G_{2n} \), where \( G_{2n} \) is the set defined in Exercise 2.3 above.

2. In this question you will show that one cannot improve the representation of Exercise 2.3 to length \( o(n \log n) \). Specifically, prove for every sufficiently large \( n \in \mathbb{N} \) there is no one-to-one function \( E : G_n \to \{0, 1\}^{\lceil 0.001n \log n \rceil + 1000} \).

**Exercise 2.5 — Multiplying in different representation.** Recall that the grade-school algorithm for multiplying two numbers requires \( O(n^2) \) operations. Suppose that instead of using decimal representation, we use one of the following representations \( R(x) \) to represent a number \( x \) between 0 and \( 10^n - 1 \). For which one of these representations you can still multiply the numbers in \( O(n^2) \) operations?

a. The standard binary representation: \( B(x) = (x_0, \ldots, x_k) \) where \( x = \sum_{i=0}^{k} x_i 2^i \) and \( k \) is the largest number s.t. \( x \geq 2^k \).

b. The reverse binary representation: \( B(x) = (x_k, \ldots, x_0) \) where \( x_i \) is defined as above for \( i = 0, \ldots, k-1 \).

c. Binary coded decimal representation: \( B(x) = (y_0, \ldots, y_{n-1}) \) where \( y_i \in \{0,1\}^4 \) represents the \( i^{th} \) decimal digit of \( x \) mapping 0 to 0000, 1 to 0001, 2 to 0010, etc. (i.e. 9 maps to 1001)

d. All of the above.

**Exercise 2.6** Suppose that \( R : \mathbb{N} \to \{0,1\}^* \) corresponds to representing a number \( x \) as a string of \( x \) 1’s, (e.g., \( R(4) = 1111, R(7) = 1111111 \), etc.). If \( x, y \) are numbers between 0 and \( 10^n - 1 \), can we still multiply \( x \) and \( y \) using \( O(n^2) \) operations if we are given them in the representation \( R(\cdot) \)?

**Exercise 2.7** Recall that if \( F \) is a one-to-one and onto function mapping elements of a finite set \( U \) into a finite set \( V \) then the sizes of \( U \) and \( V \) are the same. Let \( B : \mathbb{N} \to \{0,1\}^* \) be the function such that for every \( x \in \mathbb{N} \), \( B(x) \) is the binary representation of \( x \).
1. Prove that \( x < 2^k \) if and only if \( |B(x)| \leq k \).

2. Use 1. to compute the size of the set \( \{ y \in \{0, 1\}^* : |y| \leq k \} \) where \( |y| \) denotes the length of the string \( y \).

3. Use 1. and 2. to prove that \( 2^k - 1 = 1 + 2 + 4 + \cdots + 2^{k-1} \).

**Exercise 2.8 — Prefix-free encoding of tuples.** Suppose that \( F : \mathbb{N} \to \{0, 1\}^* \) is a one-to-one function that is prefix-free in the sense that there is no \( a \neq b \) s.t. \( F(a) \) is a prefix of \( F(b) \).

a. Prove that \( F_2 : \mathbb{N} \times \mathbb{N} \to \{0, 1\}^* \), defined as \( F_2(a, b) = F(a)F(b) \) (i.e., the concatenation of \( F(a) \) and \( F(b) \)) is a one-to-one function.

b. Prove that \( F_* : \mathbb{N}^* \to \{0, 1\}^* \) defined as \( F_*(a_1, \ldots, a_k) = F(a_1) \cdots F(a_k) \) is a one-to-one function, where \( \mathbb{N}^* \) denotes the set of all finite-length lists of natural numbers.

**Exercise 2.9 — More efficient prefix-free transformation.** Suppose that \( F : O \to \{0, 1\}^* \) is some (not necessarily prefix-free) representation of the objects in the set \( O \), and \( G : \mathbb{N} \to \{0, 1\}^* \) is a prefix-free representation of the natural numbers. Define \( F'(o) = G(|F(o)|)F(o) \) (i.e., the concatenation of the representation of the length \( F(o) \) and \( F(o) \)).

a. Prove that \( F' \) is a prefix-free representation of \( O \).

b. Show that we can transform any representation to a prefix-free one by a modification that takes a \( k \) bit string into a string of length at most \( k + O(\log k) \).

c. Show that we can transform any representation to a prefix-free one by a modification that takes a \( k \) bit string into a string of length at most \( k + \log k + O(\log \log k) \).\(^{25}\) Hint: Think recursively how to represent the length of the string.

**Exercise 2.10 — Kraft’s Inequality.** Suppose that \( S \subseteq \{0, 1\}^n \) is some finite prefix-free set.

a. For every \( k \leq n \) and length-\( k \) string \( x \in S \), let \( L(x) \subseteq \{0, 1\}^n \) denote all the length-\( n \) strings whose first \( k \) bits are \( x_0, \ldots, x_{k-1} \). Prove that 
(1) \( |L(x)| = 2^{n-|x|} \) and (2) If \( x \neq x' \) then \( L(x) \) is disjoint from \( L(x') \).

b. Prove that \( \sum_{x \in S} 2^{-|x|} \leq 1 \).
c. Prove that there is no prefix-free encoding of strings with less than logarithmic overhead. That is, prove that there is no function \( PF : \{0, 1\}^* \to \{0, 1\}^* \) s.t. \( |PF(x)| \leq |x| + 0.9 \log |x| \) for every \( x \in \{0, 1\}^* \) and such that the set \( \{PF(x) : x \in \{0, 1\}^*\} \) is prefix-free. The factor 0.9 is arbitrary; all that matters is that it is less than 1.

**Exercise 2.11 — Composition of one-to-one functions.** Prove that for every two one-to-one functions \( F : S \to T \) and \( G : T \to U \), the function \( H : S \to U \) defined as \( H(x) = G(F(x)) \) is one to one.

**Exercise 2.12 — Natural numbers and strings.** 1. We have shown that the natural numbers can be represented as strings. Prove that the other direction holds as well: that there is a one-to-one map \( StN : \{0, 1\}^* \to \mathbb{N} \). (\( StN \) stands for “strings to numbers.”)

2. Recall that Cantor proved that there is no one-to-one map \( RtN : \mathbb{R} \to \mathbb{N} \). Show that Cantor’s result implies Theorem 2.6.

**Exercise 2.13 — Map lists of integers to a number.** Recall that for every set \( S \), the set \( S^* \) is defined as the set of all finite sequences of members of \( S \) (i.e., \( S^* = \{(x_0, \ldots, x_{n-1}) \mid n \in \mathbb{N}, \forall i \leq n |x_i| \in S\} \)). Prove that there is a one-one-map from \( \mathbb{Z}^* \) to \( \mathbb{N} \) where \( Z \) is the set of \( \{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\} \) of all integers.

### 2.7 Bibliographical Notes

The study of representing data as strings mostly follows under the purview of information theory, as covered in the classic textbook of Cover and Thomas [CT06]. Representations are also studied in the field of data structures design, as covered in texts such as [Cor+09]. The two’s complement representation of signed integers was suggested in von Neumann’s classic report [Neu45] that detailed the design approaches for a stored-program computer, though similar representations have been used even earlier in abacus and other mechanical computation devices.

The idea that we should separate the definition or specification of a function from its implementation or computation might seem “obvious,” but it took quite a lot of time for mathematicians to arrive at this viewpoint. Historically, a function \( F \) was identified by rules or formulas showing how to derive the output from the input. As we discuss in greater depth in Chapter 8, in the 1800s this somewhat informal notion of a function started “breaking at the seams,” and eventually mathematicians arrived at the more rigorous definition of a function.
as an arbitrary assignment of input to outputs. While many functions may be described (or computed) by one or more formulas, today we do not consider that to be an essential property of functions, and also allow functions that do not correspond to any “nice” formula.

We have mentioned that all representations of the real numbers are inherently approximate. Thus an important endeavor is to understand what guarantees we can offer on the approximation quality of the output of an algorithm, as a function of the approximation quality of the inputs. This question is known as the question of determining the numerical stability of given equations. The Floating Points Guide website contains an extensive description of the floating point representation, as well the many ways in which it could subtly fail, see also the website 0.30000000000000004.com.

Dauben [Dau90] gives a biography of Cantor with emphasis on the development of his mathematical ideas. [Hal60] is a classic textbook on set theory, including also Cantor’s theorem. Cantor’s Theorem is also covered in many texts on discrete mathematics, including [LehmanLeightonMeyer; LewisZax19].

The adjacency matrix representation of graphs is not merely a convenient way to map a graph into a binary string, but it turns out that many natural notions and operations on matrices are useful for graphs as well. (For example, Google’s PageRank algorithm relies on this viewpoint.) The notes of Spielman’s course are an excellent source for this area, known as spectral graph theory. We will return to this view much later in this book when we talk about random walks.

Gromov’s and Pomerantz’s quotes are taken from Doron Zeilberger’s page.
FINITE COMPUTATION