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

> *"I found that every number, which may be expressed from one to ten, surpasses the preceding by one unit: afterwards the ten is doubled or tripled . . . until a hundred; then the hundred is doubled and tripled in the same manner as the units and the tens . . . and so forth to the utmost limit of numeration."*, Muhammad ibn Mūsā al-Khwārizmī, 820, translation by Fredric Rosen, 1831.

> *"A mathematician would hardly call a correspondence between the set of 64 triples of four units and a set of twenty other units,"universal", while such correspondence is, probably, the most fundamental general feature of life on Earth"*, Misha Gromov, 2013

To a first approximation, computation can be thought of as a process that maps an *input* to an *output*.

When discussing computation, it is important 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.

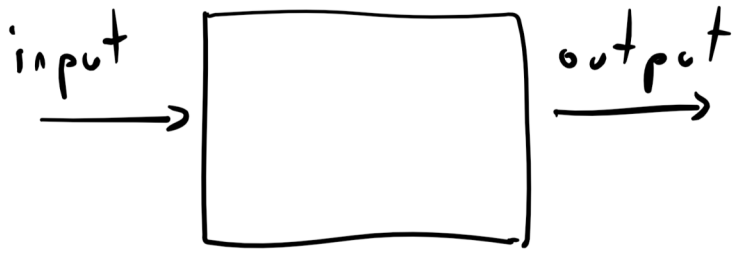

Figure 2.1: Our basic notion of *computation* is some process that maps an input to an output

In this lecture we focus on the **what** part, namely defining computational tasks. For starters, we need to define the inputs and outputs. A priori this seems nontrivial, since computation today is applied to a huge 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.

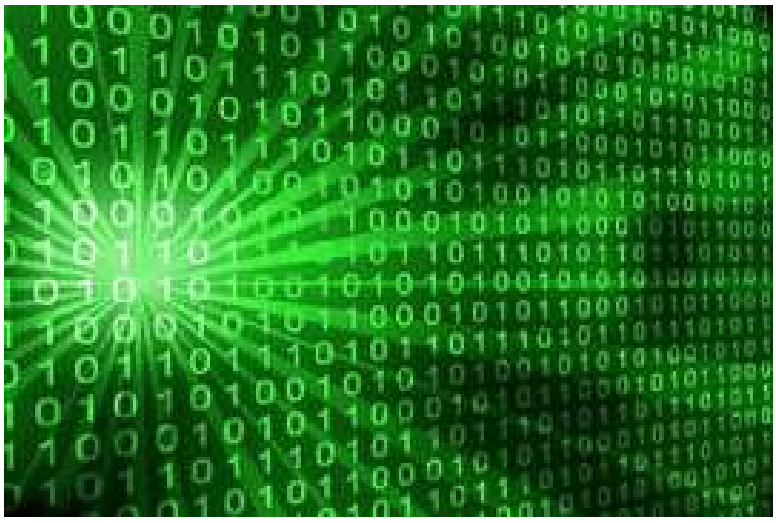

Figure 2.2: We represent numbers, texts, images, networks and many other objects using strings of zeroes and ones. Writing the zeroes and ones themselves in green font over a black background is optional.

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 a deep insight with significant implications. Many animals can convey a particular fear or desire, but what's 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.

Over the last several decades, we've seen a revolution in what we are able to represent and convey in digital form. We can capture experiences with almost perfect fidelity, and disseminate it essentially instantaneously to an unlimited audience. What's more, 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 this 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`).[1]

[1] 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.

In later lectures, we will often fall back on taking this representation for granted, and hence write something like "program P takes x as input" when x might be a number, a vector, a graph, or any other objects, when we really mean that P takes as input the *representation* of x as a binary string. However, in this lecture, let us dwell a little bit on how such representations can be devised.

2.1 Examples of binary representations

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, let us start by describing some simple representations for various natural objects.

2.1.1 Representing natural numbers

Perhaps the simplest object we want to represent is a *natural number*. That is, a member x of the set $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. We can represent a number $x \in \mathbb{N}$ as a string using the *binary basis*. Specifically, every natural number x can be written in a unique way as $x = x_0 2^0 + x_1 2^1 + \cdots + x_{n-1} 2^{n-1}$ (or $\sum_{i=0}^{n-1} x_i 2^i$ for short) where $x_0, \ldots, x_{n-1}$ are zero/one and n is the smallest number such that $2^n > x$ (and hence $x_{n-1} = 1$ for every nonzero x). We can then represent x as the string $(x_0, x_1, \ldots, x_{n-1})$.[2] For example, the number 35 is represented as the string $(1, 1, 0, 0, 0, 1)$.[3]

[2] We can represent the number zero either as some string that contains only zeroes, or as the empty string. The choice will not make any difference for us.

[3] Typically when people write down the binary representation, they would print the string x in *reverse order*, with the least significant digit as the rightmost one. Representing the number x as $(x_{n-1}, x_{n-2}, \ldots, x_0)$ will of course work just as well. We chose the particular representation above for the sake of simplicity, so the the i-th bit corresponds to 2^i, but such low level choices will not make a difference in this course. A related, but not identical, distinction is the Big Endian vs Little Endian representation for integers in computing architecture.

We can think of a representation as consisting of *encoding* and *decoding* functions. In the case of the *binary representation* for integers,

the *encoding* function $E : \mathbb{N} \to \{0,1\}^*$ maps a natural number to the string representing it, and the *decoding* function $D : \{0,1\}^* \to \mathbb{N}$ maps a string into the number it represents (i.e., $D(x_0, \ldots, x_{n-1}) = 2^0 x_0 + 2^1 x_1 + \ldots + 2^{n-1} x_{n-1}$ for every $x_0, \ldots, x_{n-1} \in \{0,1\}$). For the representation to be well defined, we need every natural number to be represented by some string, where two distinct numbers must have distinct representations. This corresponds to requiring the *encoding* function to be one-to-one, and the *decoding* function to be *onto*.

If you don't remember the definitions of *one-to-one*, *onto*, *total* and *partial* functions, now would be an excellent time to review them. Make sure you understand *why* the function E described above is one-to-one, and the function D is *onto*.

2.1.2 *Representing (potentially negative) integers*

Now that we can represent natural numbers, 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. So, the string $(\sigma, x_0, \ldots, x_{n-1}) \in \{0,1\}^{n+1}$ will represent the number

$$(-1)^{\sigma} \left[x_0 2^0 + \cdots x_{n-1} 2^n \right] \tag{2.1}$$

The decoding function of a representation should always be *onto*, since every object must be represented by some string. However, it does not always have to be one to one. For example, in this particular representation 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 $E : \mathbb{Z} \to \{0,1\}^*$ 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 two distinct integers have distinct representations.

Interpretation and context: Given a string $x \in \{0,1\}^*$, 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 x

is "supposed" to be an integer, how do we know what representation scheme it uses? The short answer is that we don't necessarily know this information, unless it is supplied from the context.[4] We can treat the same string x 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 x", we will assume that we are fixing some canonical representation scheme such as the ones above. The choice of the particular representation scheme will almost never matter, except that we want to make sure to stick with the same one for consistency.

[4] In programming language, the compiler or interpreter determines the representation of the sequence of bits corresponding to a variable based on the variable's *type*.

2.1.3 *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, simply concatenating the representations of a and b will not work.[5] For example, recall that we represent 4 as $(0,1)$ and 35 as $(1,1,0,0,0,1)$, but the concatenation $(0,1,1,1,0,0,0,1)$ of these strings is also the concatenation of the representation $(0,1,1)$ of 6 and the representation $(1,0,0,0,1)$ of 17. Hence, if we used such simple concatenation then we would not be able to tell if the string $(0,1,1,1,0,0,0,1)$ is supposed to represent 4/35 or 6/17.

[5] Recall that the *concatenation* of two strings x and y is the string of length $|x|+|y|$ obtained by writing y after x.

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 simply use a separator such as the semicolon symbol to represent, for example, the pair consisting of the numbers represented by $(0,1)$ and $(1,1,0,0,0,1)$ as the length-9 string s `01;110001`. By adding a little redundancy, We can do just that in the digital domain. The idea is that we will 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. In particular, if apply this map to every symbol of the length-9 string s above, we get the length 18 binary string 001101111100000011. More generally, the above encoding yields a one-to-one map E from strings over the alphabet Σ to binary string, such that for every $s \in \Sigma^*$, $|E(s)| = 2|s|$.

Using this, we get get a one to one map $E' : (\{0,1\}^*) \times (\{0,1\}^*) \rightarrow \{0,1\}^*$ mapping *pairs* of binary strings into a single binary string. Given every pair (a,b) of binary strings, we will first map it in a one-to way to a string $s \in \Sigma^*$ using `;` as a separator, and then map s to a single (longer) binary string using the encoding E. The same idea can be used to represent triples, quadruples, and generally all tuples of

strings as a single string (can you see why?).

2.1.4 *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 π or e. Every real number can be approximated by a rational number, and so up to a small error we can represent every real number x by a rational number a/b that is very close to x. For example, we can represent π by $22/7$ with an error of about 10^{-3} and if we wanted smaller error (e.g., about 10^{-4}) then we can use $311/99$ and so on and so forth.

This is a fine representation though a more common choice to represent real numbers is the *floating point* representation, where we represent x by the pair (a, b) of (positive or negative) integers of some prescribed sizes (determined by the desired accuracy) such that $a \times 2^b$ is closest to x.[6] The reader might be (rightly) worried about this issue of approximation. 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. This representation is called "floating point" because we can think of the number a as specifying a sequence of binary digits, and b 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. 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.

[6] You can think of this as related to scientific notation. In scientific notation we represent a number y as $a \times 10^b$ for integers a, b. Sometimes we write this as $y = a\mathtt{E}b$. For example, in many programming languages `1.21E2` is the same as `121.0`. In scientific notation, to represent π up to accuracy 10^{-3} we will simply use 3141×10^{-3} and to represent it up to accuracy 10^{-4} we will use 31415×10^{-4}.

2.1.5 *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 says this cannot be done

Theorem 2.1 — Reals are uncountable. There is no one-to-one function $RtS : \mathbb{R} \to \{0,1\}^*$. [7]

Theorem 2.1 was proven by Georg Cantor in 1874.[8] The result

[7] *RtS* stands for "reals to strings".

[8] Cantor used the set $\mathbb{N}$ rather than $\{0,1\}^*$, but one can show that these two result are equivalent using the one-to-one maps between those two sets, see Exercise 2.9. Saying that there is no one-to-one map from $\mathbb{R}$ to $\mathbb{N}$ is equivalent to saying that there is no onto map $NtR : \mathbb{N} \to \mathbb{R}$ or, in other words, that there is no way to "count" all the real numbers as $NtR(0), NtR(1), NtR(2), \ldots$. For this reason Theorem 2.1 is known as the *uncountability of the reals*.

(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 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 we will see later in this course, Cantor's ideas also play a huge role in the theory of computation.

Now that we discussed the theorem's importance, let us see the proof. Theorem 2.1 follows from the following two results:

Lemma 2.2 Let $\{0,1\}^\infty$ be the set $\{f \mid f : \mathbb{N} \to \{0,1\}\}$ of functions from $\mathbb{N}$ to $\{0,1\}$.[9] Then there is no one-to-one map $FtS : \{0,1\}^\infty \to \{0,1\}^*$.[10]

Lemma 2.3 There *does* exist a one-to-one map $FtR : \{0,1\}^\infty \to \mathbb{R}$.[11]

[9] 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 $(f(0), f(1), f(2), \ldots)$.

[10] FtS stands for "functions to strings".

[11] FtR stands for "functions to reals."

Lemma 2.2 and Lemma 2.3 together imply Theorem 2.1. To see why, suppose, towards the sake of contradiction, that there did exist a one-to-one function $RtS : \mathbb{R} \to \{0,1\}^*$. By Lemma 2.3, there exists a one-to-one function $FtR : \{0,1\}^\infty \to \mathbb{R}$. Thus, under this assumption, since the composition of two one-to-one functions is one-to-one (see Exercise 2.8), the function $FtS : \{0,1\}^\infty \to \{0,1\}^*$ defined as $FtS(f) = RtS(FtR(f))$ will be one to one, contradicting Lemma 2.2.

Now all that is left is to prove these two lemmas. We start by proving Lemma 2.2 which is really the heart of Theorem 2.1.

Proof. Let us assume, towards the sake of contradiction, that there exists a one-to-one function $FtS : \{0,1\}^\infty \to \{0,1\}^*$. Then, there is an *onto* function $StF : \{0,1\}^* \to \{0,1\}^\infty$ (e.g., see Lemma 0.1). We will derive a contradiction by coming up with some function $f^* : \mathbb{N} \to \{0,1\}$ such that $f^* \neq StF(x)$ for every $x \in \{0,1\}^*$.

The argument for this is short but subtle. We need to construct some function $f^* : \mathbb{N} \to \{0,1\}$ such that for every $x \in \{0,1\}^*$, if we let $g = StF(x)$ then $g \neq f^*$. Since two functions are identical if and only if they agree on every input, to do this we need to show that there is *some* $n \in \mathbb{N}$ such that $f^*(n) \neq g(n)$. (All these quantifiers can

be confusing, so let's again recap where we are and where we want to get to. We assumed by contradiction there is a one-to-one FtS and hence an onto StF. To get our desired contradiction we need to show the *existence* of a single f^* such that for *every* $x \in \{0,1\}^*$ there *exists* $n \in \mathbb{N}$ on which f^* and $g = StF(x)$ disagree.)

The idea is to construct f^* iteratively: for every $x \in \{0,1\}^*$ we will "ruin" f^* in one input $n(x) \in \mathbb{N}$ to ensure that $f^*(n(x)) \neq g(n(x))$ where $g = StF(x)$. If we are successful then this would ensure that $f^* \neq STF(x)$ for every x. Specifically, for every $x \in \{0,1\}^*$, let $n(x) \in N$ be the number $x_0 + 2x_1 + 4x_2 + \cdots + 2^{k-1}x_{k-1} + 2^k$ where $k = |x|$. That is, $n(x) = 2^k + \sum_{i=0}^{k-1} 2^i x_i$. If $x \neq x'$ then $n(x) \neq n(x')$ (we leave verifying this as an exercise to you, the reader).

Now for every $x \in \{0,1\}^*$, we define

$$f^*(n(x)) = 1 - g(n(x)) \tag{2.2}$$

where $g = StF(x)$. For every n that is not of the form $n = n(x)$ for some x, we set $f^*(n) = 0$. Eq. (2.2) is well defined since the map $x \mapsto n(x)$ is one-to-one and hence we will not try to give $f^*(n)$ two different values.

Now by Eq. (2.2), for every $x \in \{0,1\}^*$, if $g = StF(x)$ and $n = n(x)$ then $f^*(n) = 1 - g(n) \neq g(n)$. Hence $StF(x) \neq f^*$ for every $x \in \{0,1\}^*$, contradicting the assumption that StF is onto. This proof 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. 2.3. ■

R **Genearlizing beyond strings and reals** Lemma 2.2 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 $F : \{0,1\}^S \to S$ where $\{0,1\}^S$ denotes the set $\{f \mid f : S \to \{0,1\}\}$ 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 2.2 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 $\{0,1\}^{\mathbb{R}}$ is "even bigger" than $\mathbb{R}$. Cantor used these ideas to construct an infinite hierarchy of shades of infinity. The number of such shades turn out to be much larger than $|\mathbb{N}|$ or even $|\mathbb{R}|$. He denoted the cardinality of

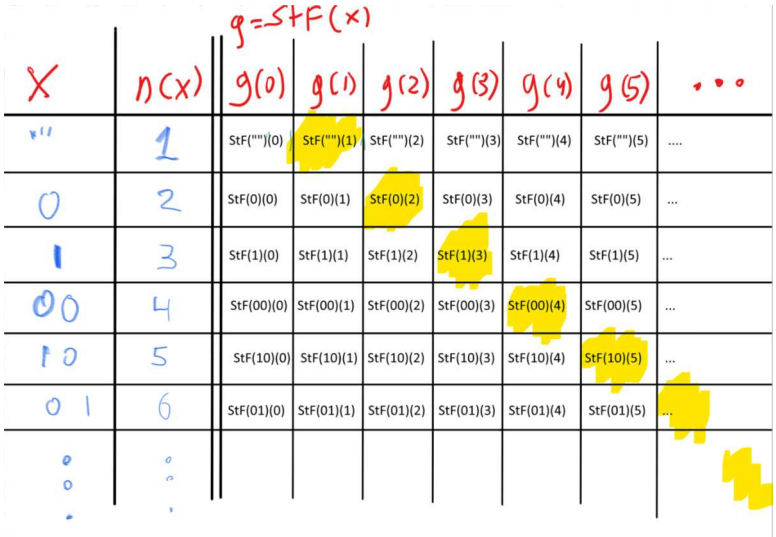

Figure 2.3: We construct a function f^* such that $f^* \neq StF(x)$ for every $x \in \{0,1\}^*$ by ensuring that $f^*(n(x)) \neq StF(x)(n(x))$ for every $x \in \{0,1\}^*$. We can think of this as building a table where the columns correspond to numbers $m \in \mathbb{N}$ and the rows correspond to $x \in \{0,1\}^*$ (sorted according to $n(x)$). If the entry in the x-th row and the m-th column corresponds to $g(m))$ where $g = StF(x)$ then f^* is obtained by going over the "diagonal" elements in this table (the entries corresponding to the x-th row and $n(x)$-th column) and enduring that $f^*(x)(n(x)) \neq StF(x)(n(x))$.

> $\mathbb{N}$ by $\aleph_0$, where $\aleph$ is the first letter in the Hebrew alphabet, and called the the next largest infinite number by $\aleph_1$. Cantor also made the continuum hypothesis that $|\mathbb{R}| = \aleph_1$. We will come back to the very interesting story of this hypothesis later on in this course. This lecture of Aaronson mentions some of these issues (see also this Berkeley CS 70 lecture).

To complete the proof of Theorem 2.1, we need to show Lemma 2.3. This requires some calculus background, but is otherwise straightforward. The idea is that we can construct a one-to-one map from $\{0,1\}^\infty$ to the real numbers by mapping the function $f : \mathbb{N} \to \{0,1\}$ 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 real series before, then the formal proof might be a little hard to follow. This part is not the core of Cantor's argument, nor are such limits very crucial to this course, so feel free to also just take Lemma 2.3 on faith and skip the formal proof.

Proof of Lemma 2.3. 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 (that we won't repeat here) that for every $f : \mathbb{N} \to \{0,1\}$, the sequence $(S(f)_n)_{n=0}^{\infty}$ has

a *limit*. That is, there exists some value x such that for every $\epsilon > 0$, if n is sufficiently large then $|S_f(n) - x| < \epsilon$. We define $FtR(f)$ to be this value x. To show that FtR is one to one, we need to show that $FtR(f) \neq FtR(g)$ for every distinct $f, g : \mathbb{N} \to \{0,1\}$. Let $f \neq g$ be such functions, and let k be the smallest number for which $f(k) \neq g(k)$. Assume without loss of generality that $f(k) = 0$ and $g(k) = 1$. Then, if $S = \sum_{i=0}^{k-1} 10^{-i} f(i) = \sum_{i=0}^{k-1} 10^{-i} f(i)$, we get that for every $n > k+1$, $S(f)_n = S + \sum_{i=k+1}^{n} 10^{-i} f(i)$ and $S(g)_n = S + 10^{-k} + \sum_{i=k+1}^{n} 10^{-i} g(i)$. Clearly, the limit $FtR(g)$ is at least $S + 10^{-k}$. On the other hand, we claim that for every $n > k+1$, $S(f)_n \leq S + 2 \cdot 10^{-k-1} < S + 10^{-k}$, and hence in particular $FtR(f) < FtR(g)$. Indeed, since $f(i) \in \{0,1\}$ for every i, $\sum_{i=k+1}^{n} f(i) 10^{-i} \leq \sum_{i=k+1}^{n} 10^{-i}$ which by formula for geometric series, equals $10^{-k-1} \frac{1-10^{-(n-k-1)}}{1-10^{-1}} \leq 10^{-k-1}/0.9 \leq 2 \cdot 10^{-k-1}$. ■

2.2 *Beyond numbers*

We can of course represent objects other than numbers as binary strings. Let us give a general definition for representation:

Definition 2.4 — String representation. Let $\mathcal{O}$ be some set. A *representation scheme* for $\mathcal{O}$ consists of a pair (E, D) where $E : \mathcal{O} \to \{0,1\}^*$ is a total one-to-one function, $D : \{0,1\}^* \to_p \mathcal{O}$ is a (possibly partial) function, and such that D and E satisfy 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 2.5 Suppose that $E : \mathcal{O} \to \{0,1\}^*$ is one-to-one. Then there exists a function $D : \{0,1\}^* \to \mathcal{O}$ such that $D(E(o)) = o$ for every $o \in \mathcal{O}$.

Proof. Let o_0 be some arbitrary element of 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 $o \in \mathcal{O}$. ■

Note that, while in general we allowed the decoding function to be *partial*. This proof shows that we can always obtain a *total* decoding function if we need to. This observation can sometimes be useful.

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

Lemma 2.6 For every two finite sets S, T, there exists a one-to-one $E : S \to T$ 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 $E : S \to T$ iff $k \leq m$. 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 $E : S \to T$ 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.[12] ■

[12] This direction is sometimes known as the "Pigeon Hole Principle": the principle that if you have pigeon coop with m holes, and $k > m$ pigeons, then there must be two pigeon in the same hole.

Now the size of $\{0,1\}^n$ is 2^n, and the size of $\{0,1\}^{\leq n}$ is only slightly bigger: $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$ by the formula for a geometric series.

2.2.2 *Prefix free encoding*

In our discussion of the representation of 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. An encoding function $E : \mathcal{O} \to \{0,1\}^*$ is *prefix free* if there are no two objects $o \neq o'$ such that the representation $E(o)$ is a *prefix* of the representation $E(o')$. The definition of prefix is as you would expect: a length n string x is a

prefix of a length $n' \geq n$ string x' if $x_i = x'_i$ for every $1 \leq i \leq n$. Given a representation scheme for $\mathcal{O}$ with a prefix-free encoding map, we can use simple concatenation to encode tuples of objects in $\mathcal{O}$:

Theorem 2.7 — Prefix free implies tuple encoding. Suppose that (E, D) is a representation scheme for $\mathcal{O}$ and E is prefix free. Then there exists a representation scheme (E', D') for $\mathcal{O}^*$ such that for every $(o_0, \ldots, o_{k-1}) \in \mathcal{O}^*$, $E'(o_0, \ldots, o_{k-1}) = E(o_0)E(o_1)\cdots E(o_{k-1})$.

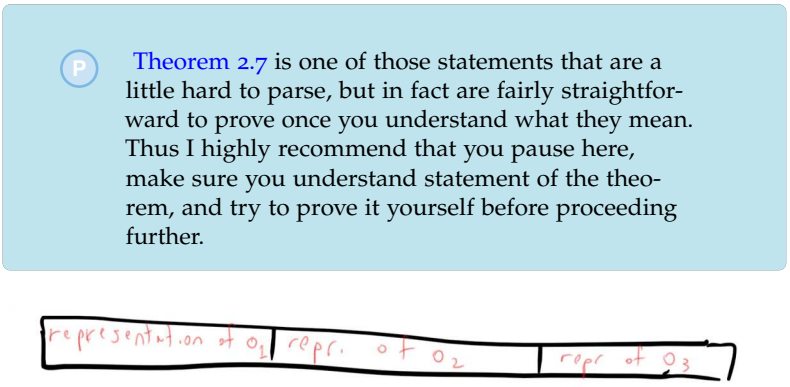

Figure 2.4: If we have a prefix-free representation of each object then we can concatenate the representations of k objects to obtain a representation for the tuple $(o_1, \ldots, o_k)$.

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 2.5). 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. We now show the formal proof.

Proof of Theorem 2.7. By Lemma 2.5, to prove the theorem it suffices show that E' is one-to-one. 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

$$E'(o_0, \ldots, o_{k-1}) = E'(o'_0, \ldots, o'_{k'-1}) \,, \tag{2.3}$$

and denote this string by x.

We denote $x_i = E(o_i)$ and $x'_i = E(o'_i)$. By our assumption and the definition of E', $x_0 x_1 \cdots x_k = x'_0 x'_1 \cdots x'_k$.

Let's make the assumption A that there is some index $i \in [\min\{k,k'\}]$ such that $o_i \neq o'_i$. Now, let i be the first such index, and hence $o_j = o'_j$ for all $j < i$ but $o_i \neq o'_i$, and so (since E is one-to-one) also $x_i \neq x'_i$. That means that if we write $p = x_0 \cdots x_{i-1}$, then by Eq. (2.3) the first $|p| + |x_i|$ bits of the string x need to equal $p\,x_i$ (i.e., the concatenation of p and x_i) and the first $|p| + |x'_i|$ bits of x needs to equal $p\,x'_i$. If $|x_i| = |x'_i|$, then the only way this can happen is if $p\,x_i = p\,x'_i$, which implies $x_i = x'_i$, in contradiction to the fact that E is one-to-one. Otherwise, without loss of generality $|x_i| > |x'_i|$,[13] and so px'_i must be a *prefix* of px_i, but this contradicts the prefix-freeness of E. The only remaining case is when Assumption A is false. Since the tuples are different, if $o_i = o'_i$ for every $i \in [\min\{k,k'\}]$, it must mean that $k \neq k'$. Without loss of generality, assume that $k < k'$ (again, check that this is justified!). But then, since $o_i = o'_i$ for all $i \in [k]$, it holds that $x_0 \cdots x_{k-1} = x_0 \cdots x_{k-1} x'_k \cdots x'_{k'-1}$. But this can only happen if x'_i for $i \geq k$ is the empty string, while a prefix-free encoding can never encode an object as the empty string (can you see why?). ■

[13] This is one of our first uses of the phrase "without loss of generality". Make sure you understand why it is justified! If you are not sure, you can try working out the case that $|x_i| > |x'_i|$ and see why the proof is symmetric.

2.2.3 *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 2.8 Let $E : \mathcal{O} \to \{0,1\}^*$ be a one-to-one function. Then there is a one-to-one prefix-free encoding $\overline{E}$ such that $|\overline{E}(o)| \leq 2|E(o)| + 2$ for every $o \in \mathcal{O}$.

P 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 yourself, using the same technique we used for representing rational numbers.

Proof of Lemma 2.8. 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 the sake of 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 $i < |y|$.)

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 $\overline{E}(o) = PF(x)$ is not a prefix of $\overline{E}(o') = PF(x')$. 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 $\overline{E}(o) = PF(x)$ is not a prefix of $\overline{E}(o') = PF(x')$. ■

In fact, we can even obtain a more efficient transformation where $|E'(o)| \leq |o| + O(\log |o|)$. We leave proving this as an exercise (see Exercise 2.6).

2.2.4 *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 simply concatenating the representation of each symbol. One such representation is the ASCII that represents 128 letters and symbols as strings of 7 bits. Since it is a fixed-length representation it is automatically prefix free (can you see why?). Unicode is a representation of (at the time of this writing) about 128,000 symbols into 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.

2.2.5 *Representing vectors, matrices, images*

Once we can represent numbers, and lists of numbers, then we can obviously represent *vectors* (which are just lists of numbers). Similarly, we can represent lists of lists and so in particular *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.[14] Thus an image of n pixels would be represented of a list of n such length-three lists. A video can be represented as a list of images.[15]

[14] We can restrict to three basic colors since (most) humans only have three types of cones in their retinas. We would have needed 16 basic colors to represent colors visible to the Mantis Shrimp.

[15] 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 course.

2.2.6 *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 $A \in \{0,1\}^{n^2}$ such that $A_{i,j} = 1$ iff the edge $\overrightarrow{i\,j} \in E$. We can transform an undirected graph to a directed graph by replacing every edge $\{i,j\}$ with both edges $\overrightarrow{i\,j}$ and $\overleftarrow{i\,j}$

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 $G = (V,E)$ a a list of n lists, where the i-th list consists of the out-neighbors of vertex i. The difference between these representations can be important for some applications, though for us would typically be immaterial.

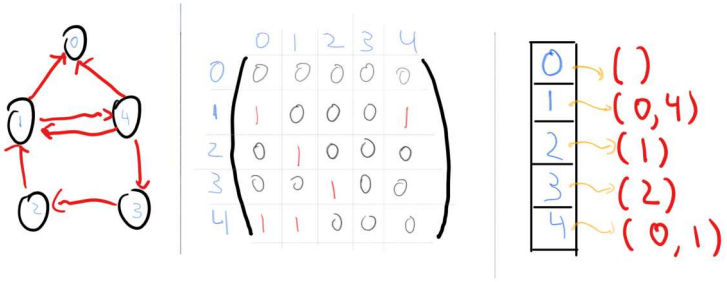

Figure 2.5: Representing the graph $G = (\{0,1,2,3,4\}, \{(1,0),(4,0),(1,4),(4,1),(2,1),(3,2),(4,3)\})$ in the adjacency matrix and adjacency list representations.

2.2.7 *Representing lists*

If we have a way of represent objects from a set $\mathcal{O}$ as binary strings, then we can represent lists of these objects ny 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 $E : \mathcal{O} \to \{0,1\}^*$, 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,[1011,00111]]"` over the alphabet Σ. By encoding every element of Σ itself as a three-bit string, we can transform any representation for objects $\mathcal{O}$ into a representation that allows to represent (potentially nested) lists of these objects.

2.2.8 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 $\underline{x}$ and let $\underline{y} = F(\underline{x})$, then the integer y represented by the string $\underline{y}$ 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 order 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.

2.3 Defining computational tasks

Abstractly, a *computational process* is some process that takes an input which is a string of bits, and produces an output which is a string of bits. This transformation of input to output can be done using a modern computer, a person following instructions, the evolution of some natural system, or any other means.

In future lectures, we will turn to mathematically defining computational process, but, as we discussed above for now we want to focus on *computational tasks*; i.e., 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. But for most of this course, 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 com-

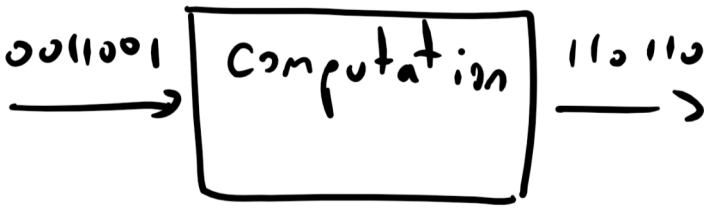

Figure 2.6: A computational process

puting a function from $\{0,1\}^*$ to $\{0,1\}^*$. We've seen that there is more than one way to solve this computational task, and in fact, we still don't 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 huge difference in their computational difficulty.
- Given the code of a Python program, is there an input that would force it into an infinite loop. This corresponds to computing a partial function from $\{0,1\}^*$ to $\{0,1\}$; though it is easy to make it into a total function by mapping every string into the trivial Python program that stops without doing anything. 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 $\{0,1\}$.

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 F.[16] Hence many texts refer to such as computational task as *deciding a language*.

[16] The language terminology is due to historical connections between the theory of computation and formal linguistics as developed by Noam Chomsky.

For every particular function F, there can be several possible *algorithms* to compute F. We will be interested in questions such as:

- For a given function F, can it be the case that *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 can't 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 *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 *algorithm*, which is what we'll do in the next lecture.

2.3.1 *Advanced note: beyond computing functions*

Functions capture quite a lot of computational tasks, but one can consider more general settings as well. For starters, we can and will talk about *partial* functions, that are not defined on all inputs. When computing a partial function, we only need to worry about the inputs on which the function is defined. Another way to say it is that we can design an algorithm for a partial function F under the assumption that someone "promised" us that all inputs x would be such that $F(x)$ is defined (as otherwise we don't care about the result). Hence such tasks are also known as *promise problems*.

Another generalization is to consider *relations* that may have more than one possible admissible output. For example, consider the task of finding any solution for a given set of equation. A *relation* R maps a string $x \in \{0,1\}^*$ 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 on in this course we will consider even more general tasks,

including *interactive* tasks, such as finding good strategy in a game, tasks defined using probabilistic notions, and others. However, for much of this course 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.

2.4 Lecture summary

- We can represent essentially every object we want to compute on using binary strings.
- A representation scheme for a set of objects $\mathcal{O}$ is a one-to-one map from $\mathcal{O}$ to $\{0,1\}^*$.
- A basic computational task is the task of *computing a function* $F : \{0,1\}^* \to \{0,1\}^*$. This encompasses not just arithmetical computations such as multiplication, factoring, etc. but a great many other tasks arising in areas as diverse as scientific computing, artificial intelligence, image processing, data mining and many many more.
- We will study the question of finding (or at least giving bounds on) what is the *best* algorithm for computing F for various interesting functions F.

2.5 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 — 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 2.3 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 2.4 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.

a. Prove that $x < 2^k$ if and only if $|B(x)| \leq k$.

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

c. Use a. and b. to prove that $2^k - 1 = 1 + 2 + 4 + \cdots + 2^{k-1}$. ■

Exercise 2.5 — 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 2.6 — 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 $k + \log k + O(\log \log k)$.[17] ■

[17] Hint: Think recursively how to represent the length of the string.

Exercise 2.7 — 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. ■

Exercise 2.8 — 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 2.9 — 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 2.1.

■

2.6 *Bibliographical notes*

The idea that we should separate the *definition* or *specification* of a function from its *implementation* or *computation* might seem "obvious", but it took some 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 our lecture on uncomputability, in the 1800's 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.

Gromov and Pomerantz's quotes are lifted from Doron Zielberger's page.

2.7 *Further explorations*

Some topics related to this lecture that might be accessible to advanced students include:

- *Succinct* data structures. These are representations that map objects from some set $\mathcal{O}$ into strings of length not much larger than the minimum of $\log_2 |\mathcal{O}|$ but still enable fast access to certain queries, see for example this paper.
- We've 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 is known as the question of numerical stability.
- The linear algebraic view of graphs. 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 this course are an excellent source for this area, known as *spectral graph theory*. We might discuss this view much later in this course when we talk about *random walks*.

2.8 *Acknowledgements*