Strings, Encodings, and Computational Problems

In general, a *computational task* involves an *input*, some kind of computing device or mechanism, and an *output*. In the real world, the input could be an object of many different types, such as an integer, a graph, an image, a list of integers, . . . We will need to *encode* all such mathematical objects as *strings*. After all, they are eventually stored as just `0`'s and `1`'s on any digital computer.

Remark. Thinking about how to encode an object as a string is a very similar to thinking about *file formats* for different types of objects (e.g., .jpg for images, or GraphML for graphs).

Note. Though encodings/file-formats are eventually boiled down to the "alphabet" of `0`'s and `1`'s on digital computers, we also often think of using larger, more human-understandable, "alphabets" like ASCII.

1 Alphabets and strings

Definition (Alphabet)**.** An *alphabet*, often denoted Σ, is a finite nonempty set of *symbols*.

Note. The most common choice in Complexity Theory is the binary alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$.

Example. Examples would include:

- $\Sigma = \{\texttt{a}, \texttt{b}\}$ (a popular choice in the study of finite automata).
- The alphabet of digits, $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}, \dots, \texttt{9}\}$.
- The alphabet of letters, $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \dots, \texttt{z}\}$.
- The following somewhat peculiar 7-symbol alphabet: $\Sigma = \{\texttt{0}, \texttt{1}, \widehat{\texttt{0}}, \widehat{\texttt{1}}, \texttt{x}, \texttt{\#}, \texttt{;}\}$.
- ASCII, a certain alphabet of size 256.

Exercise. Which of the following is a valid alphabet?

- $\Sigma_0 = \emptyset$.
- $\Sigma_1 = \{\texttt{a}, \texttt{2}, \texttt{\$}\}$.
- $\Sigma_2 = \mathbb{N}$, the natural numbers.

Solution. Only Σ_1 is a valid alphabet; alphabets cannot be empty or infinite. ■

Note. An alphabet *may* have size 1, although in practice this is sort of "discouraged". This is called a *unary* alphabet, and the most popular way of writing it is $\Sigma = \{\texttt{1}\}$. We can in fact always encode objects with unary alphabets, but — unlike with binary encodings — this is *highly* inefficient.

Definition (String)**.** A *string* over alphabet Σ is any finite sequence of symbols from Σ.

Definition (String length)**.** The *length* of a string x, denoted $|x|$, is the number of symbols in it.

Definition (The empty string)**.** A string of length zero is always allowed; this is called the *empty string* and is usually written as ϵ.

Note. In this course we do *not* allow strings of infinite length, unless otherwise specified.

Definition (Σ^n)**.** If Σ is an alphabet and $n \in \mathbb{N}$, then Σ^n denotes the set of strings over Σ of length exactly n.

Example. $\{\texttt{0}, \texttt{1}\}^2 = \{\texttt{00}, \texttt{01}, \texttt{10}, \texttt{11}\}$.

Exercise. Let Σ be an alphabet with k symbols. What is a formula for $|\Sigma^n|$, the number of strings over Σ of length n? (Be careful about $n = 0$.)

Solution. The answer is k^n. This formula holds perfectly well for $n = 0$; the comment about "being careful" was a bit of a trick. ∎

Remark. If Σ is an alphabet, then $\Sigma^0 = \{\epsilon\}$, and Σ^1 is the same as Σ.

Definition (Σ^*)**.** If Σ is an alphabet, then Σ^* denotes the set of *all* strings (of finite length) over Σ. In other words, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$.

Note. Σ^* is always an *infinite* set, but every element (string) in Σ^* is of *finite* length.

2 Encodings

Now we want to define exactly how mathematical objects are encoded by strings. There's a catch though, which is that we don't want to have to pedantically go through every possible mathematical object and specify some rigorous "file format" (encoding) for it. It would just be too much trouble. What we're going to do instead is *pretend* that we've done that. We *pretend* that for every type of mathematical object T, and also for every alphabet Σ, there is some reasonable, fixed, "ISO standard" way of encoding an object X of type T as a string over alphabet Σ. This pretense makes the following definition a little unrigorous, but we prefer it to over-elaborate specifications.

Definition (Encodings)**.** If X is a mathematical object, and Σ is an alphabet, then $\langle X \rangle_\Sigma$ denotes "the" *encoding* of X as a string in Σ^* using a fixed, "reasonable" scheme. If we just write $\langle X \rangle$ without specifying Σ, it means that $\Sigma = \{\texttt{0}, \texttt{1}\}$.

Example (Encoding natural numbers)**.** Let's discuss encoding the natural numbers $\mathbb{N}$. The standard choice here is that $\langle X\rangle$ (meaning $\langle X\rangle_{\{\mathtt{0},\mathtt{1}\}}$) is "$X$ written in base-2". Further examples:

- If $\Sigma = \{\mathtt{0},\mathtt{1}\}$, then $\langle 13\rangle_\Sigma = \mathtt{1101} \in \Sigma^4$.
- If $\Sigma = \{\mathtt{0},\mathtt{1},\mathtt{2},\dots,\mathtt{9}\}$, then $\langle 13\rangle_\Sigma = \mathtt{13} \in \Sigma^2$. In general, writing numbers in base-10 uses about $3\times$ fewer symbols then writing them in base-2, but in Complexity Theory we rarely get worked up about "constant factors", so we don't consider it a big deal whether numbers are encoded in base-10 or base-2.
- If $\Sigma = \{\mathtt{1}\}$, then $\langle 13\rangle_\Sigma = \mathtt{1111111111111}$ (thirteen consecutive 1's). This is bad. Encoding numbers in unary ("base-1") uses *exponentially* more symbols than encoding them in binary (or any higher base). This is why we don't consider using a unary alphabet to be "reasonable".

Example (Encoding images)**.** Say we want to encode the class of 1000×1000 pixel black-and-white images. If X is such an image, then $\langle X\rangle$ would be a string in $\{\mathtt{0},\mathtt{1}\}^{1{,}000{,}000}$. We would still need to specify whether $\mathtt{0}$ = black and $\mathtt{1}$ = white, or the other way around; and also, whether we list the pixel colors by rows or by columns. Any of these choices would be reasonable, and we leave it underspecified unless it's really essential to know.

Example (Encoding pairs using punctuation)**.** Suppose we want need to encode *ordered pairs* of natural numbers; for example, $X = (5,2)$. One perfectly reasonable way to do this involves taking the alphabet you usually use to encode numbers, $\{\mathtt{0},\mathtt{1}\}$, and *enlarging* it with "punctuation" symbol, e.g., #. Thus with alphabet $\Sigma = \{\mathtt{0},\mathtt{1},\mathtt{\#}\}$, a reasonable encoding for $X = (5,2)$ would be $\langle X\rangle_\Sigma = \mathtt{101\#10}$.

Note. For whatever reason, the symbol # came to be the most common "punctuation" mark used in Complexity Theory. In the olden days it was endlessly debated whether this symbol should be called "sharp" or "pound" or "number" — but now we can all happily call it "hashtag".

Example (Encoding pairs without punctuation)**.** Suppose we again want to encode ordered pairs of natural numbers, but we want to stick with the alphabet $\{\mathtt{0},\mathtt{1}\}$. There are several "reasonable" ways to do this, and again we prefer not to overzealously insist on one particular choice. The simplest and most generic idea is to just take the scheme from Example (Encoding pairs using punctuation) using the punctuation mark #, and then to "re-encode" each symbol from $\Sigma_3 = \{\mathtt{0},\mathtt{1},\mathtt{\#}\}$ using two bits. For example, we could encode $\mathtt{0} \in \Sigma_3$ as 00, and $\mathtt{1} \in \Sigma_3$ as 11, and $\mathtt{\#} \in \Sigma_3$ as 01. The sequence 10 is not used here, but that's fine. Under this system, we would have

$$\langle (5,2)\rangle_{\{\mathtt{0},\mathtt{1}\}} = \mathtt{110011011100}.$$

In this system, the number of symbols needed for $\langle (a,b)\rangle$ is $2|\langle a\rangle| + 2|\langle b\rangle| + 2$.

Exercise. Not that we'll care much about losing factors of 2, but... Come up with an alternative scheme for encoding pairs of natural numbers over the alphabet $\{0, 1\}$ with the property that $|\langle(a, b)\rangle|$ is roughly $2|\langle a\rangle| + |\langle b\rangle|$, meaning that we only lose a factor of 2 on *one* of the two numbers. Can you do even better (losing no factors of 2 at all)?

Hint. "$\texttt{1}^{|\langle a\rangle|}\texttt{0}\langle a\rangle\langle b\rangle$"

Regarding what counts as a "reasonable" encoding, the following definition is not fully rigorous, but hopefully it gets the idea across:

Definition (Reasonable encoding)**.** A *reasonable encoding scheme* should have the following properties:

- If $X \neq Y$, then $\langle X\rangle \neq \langle Y\rangle$. That is, the encoding $\langle\cdot\rangle$ is injective.
- There are "simple" and "efficient" algorithms that: (i) take as input the string-encoding $\langle X\rangle$ of an object X, and produce a data structure storing X; (ii) conversely, take a data structure storing X and output the string-encoding $\langle X\rangle$.
- The length of $\langle X\rangle$ in symbols is "not much longer" than it "needs to be".

Note. *Most* of the time, the precise details of how we encode objects by strings is not very important for Algorithms and Complexity Theory. However, on those occasions when it *does* make a difference, one should always explicitly specify the encoding. For example, in analyzing the running time of various graph algorithms, it can make a difference whether one is encoding X in "adjacency matrix" format, or "adjacency list" format, or something else. And when analyzing Turing Machine algorithms operating on natural numbers, it can sometimes make some difference whether the numbers are encoded with the most significant bit on the left ("big-endian") or with the least significant bit on the left ("little-endian"). Again, if it ever makes a notable difference, one should specify encoding details.

It will also sometimes be convenient to have notation for "decoding" — i.e., converting a string back to a mathematical object. Unlike the $\langle\cdot\rangle$ notation for encoding, there is no "standard" notation for decoding. So we first need to introduce:

Definition (455 Slang)**.** The phrase *455 Slang* refers to nonstandard terminology, notation, in-jokes, etc. that are specific to this Undergraduate Complexity Theory class at Carnegie Mellon. Be careful that if you ever use 455 Slang outside of class, people will not know you mean, so you'll have to explain!

Definition (Decoding)**.** Let T be a type of mathematical object — e.g., natural number, graph, order-pair-of-integers, etc. Given a string $w \in \Sigma^*$, we write $[w]_T$ for the mathematical object X of type T whose encoding over Σ is w; that is, the X such that $\langle X\rangle_\Sigma = w$.

Important. The decoding notation $[\cdot]_T$ is **455 Slang**. You'll have to explain this notation if you use it outside class.

Example. $[\texttt{110011100}]_{\mathbb{N}} = 412$.

Our definition of "decoding" skirts around an issue that may have occurred to you: what to do if you're asked to "decode" a string which is not a valid encoding? E.g., in Example (Encoding pairs using punctuation), a string like `###00` can't be parsed as a valid encoding of an ordered pair of natural numbers. If an algorithm that expects as input (the encoding of) an order pair, but the input string it gets is the "garbage" string `###00`, what should happen? There are several acceptable conventions here, but we'll find it most convenient to set things up so that *every possible input string is allowed*. To that end:

Definition (GUIDO)**.** (Needless to say, this is **455 Slang**.) The *GUIDO* convention is:

Garbage/**U**nparsable **I**nputs = **D**efault **O**bject.

Do **not** send me inputs like this.

The meaning of this is that for every type T of mathematical object, we pick some simple "default object" O, and then any "invalid" string is defined to decode to O. For example, the "default" object for natural numbers could be 0, and the "default" object for order pairs of natural numbers could be $(0, 0)$.

Example. Returning to Example (Encoding pairs using punctuation), where $\langle(5, 2)\rangle_{\{\texttt{0},\texttt{1},\texttt{\#}\}} = \texttt{101\#10}$, according to GUIDO we would have the decoding $[\texttt{\#\#\#00}] = (0, 0)$.

Note. Under GUIDO, some mathematical objects may have more than one encoding. *This is perfectly fine.* In the previous example, both the string `0#0` and `##` represent the pair $(0, 0)$.

Remark. Although it seems like there's lots of technicalities in this chapter, try not to get hung up on them too much. The main takeaways are:

- We typically encode things using the alphabet $\{\texttt{0}, \texttt{1}\}$, and natural numbers especially are typically encoded in base-2.

- It's perfectly fine to throw punctuation marks and other symbols into your alphabet for convenience.
- The GUIDO convention is chosen to allow for the following convenience: every string in Σ^* is technically an "allowable" input.

3 Computational tasks and problems

Having formalized encoding and decoding for inputs/outputs, the next step is formalizing computation. This comes in two parts. First we formalize the notion of a computational task to be solved; then we formalize the model for algorithms/computation itself. In this section we will define three kinds of computational tasks: *decision problems*, *function problems*, and *search problems*.

Definition (Decision problem)**.** A *decision problem* is the following kind of computational task:

- *Input:* a string.
- *Output:* yes/no.

Mathematically, it is represented by a function $f : \Sigma^* \to \{\text{no}, \text{yes}\}$. We also often use the representation $\texttt{0} = \text{no}$ and $\texttt{1} = \text{yes}$.

Example (IsPrime)**.** Informally, the **IsPrime** decision problem is the task of determining whether an input number is prime or not. More formally, it would be the function **IsPrime** : $\{\texttt{0}, \texttt{1}\}^* \to \{\text{no}, \text{yes}\}$ with

$$\textbf{IsPrime}(\texttt{0}) = \text{no},\ \textbf{IsPrime}(\texttt{1}) = \text{no},\ \textbf{IsPrime}(\texttt{10}) = \text{yes},$$
$$\textbf{IsPrime}(\texttt{11}) = \text{yes},\ \textbf{IsPrime}(\texttt{100}) = \text{no},$$

etc.

Example (ExistsPath)**.** Informally, the **ExistsPath** decision problem is the following: Given as input is a graph G and two vertices s and t. The task is to determine if there is a path from s to t in G.

More formally, this would be the function **ExistsPath** : $\{\texttt{0}, \texttt{1}\}^* \to \{\text{no}, \text{yes}\}$ which maps $\langle G, s, t\rangle$ (the encoding of a triple: undirected graph G, vertex s, vertex t) to yes or no depending on whether or not there is a path from s to t in G.

Remark. Here is an example of why the GUIDO principle (Definition (GUIDO)) is convenient; it means that decision problems always have Σ^* as their whole domain. In the **ExistsPath** example, we might have the convention that a "garbage/unparsable" string, one that is not a valid encoding of a (G, s, t) triple, is interpreted as

$$G = \text{the graph with one vertex called } 1\text{, and no edges,} \quad s = t = 1.$$

In this case, the correct output happens to be "yes", since technically in a 1-vertex graph, there is a path from that vertex to itself. Again, you shouldn't get too worried about technicalities; the idea here is just that our convention means that every string is a valid input.

Definition (Function problem)**.** A *function problem* is the following kind of computational task:

- *Input:* a string.
- *Output:* the unique "answer" (encoded as a string).

Mathematically, it is represented by a function $f : \Sigma^* \to \Sigma^*$.

Example (DecimalToBinary)**. DecimalToBinary** : $\{\mathtt{0}, \mathtt{1}, \dots, \mathtt{9}\}^* \to \{\mathtt{0}, \mathtt{1}\}^*$ is a function problem, namely, the task of converting a natural number written in base-10 to its base-2 representation. For example, **DecimalToBinary**$(\mathtt{13}) = \mathtt{1101}$.

Example (PrimeFactorization)**. PrimeFactorization** : $\{\mathtt{0}, \mathtt{1}\}^* \to \{\mathtt{0}, \mathtt{1}\}^*$ is a function problem, namely, the task of determining the prime factorization of a given natural number. (Here the input would be encoded in binary, and the output would be an encoding of a list of prime numbers. One would have to make some convention for what the correct answer for **PrimeFactorization**$(\mathtt{0})$ is.)

Definition (Search problem)**.** A *search problem* is similar to a function problem except that for each input, the "answer" might not be unique, or there might be no answer. The task is to output any correct answer when there is one, or (the encoding of) "no solution" when there is none.

Example (PrimeFactor)**. PrimeFactor** is the following search problem: The input is $\langle x \rangle$ for $x \in \mathbb{N}$, and the output should be $\langle p \rangle$ for any prime p dividing x.

Example (FindPath)**. FindPath** is the following search problem: The input is $\langle G, s, t \rangle$ where G is a graph and s and t are vertices in G. The output should be (the encoding of) any path from s to t in G if one exists, or (the encoding of) "not connected" if there is no path from s to t in G.

Exercise. Formally define each of the following tasks: **Palindrome** (check if a string is a palindrome), **PatternMatch** (find an occurrence of the first input string in the second input string), **Reverse** (reverse a string).

Hint. **Palindrome** should be a decision problem; **PatternMatch** should be a search problem; **Reverse** should be a function problem.

Even though search/function problems are probably more common in "real life", in Complexity Theory we more often focus on *decision problems*. There are two reasons for this. One is simplicity. The second is that it is usually "without (much) loss of generality". We will discuss this more later, but often there is a reasonably efficient "reduction" from search/function problems to decision problems.

Example (ExistsPrimeFactorBetween vs FindPrimeFactorBetween)**.** Consider the following two tasks:

- The decision problem **ExistsPrimeFactorBetween**: Given as input natural numbers x, *lo*, *hi*, does x have a prime factor between *lo* and *hi*?
- The search problem **FindPrimeFactorBetween**: Given as input natural numbers x, *lo*, *hi*, output a prime factor of x between *lo* and *hi*, if one exists.

Suppose, as suggested, that we focus mainly on algorithms for the decision problem **ExistsPrimeFactorBetween**. Let A be such an algorithm. Now suppose we actually want an algorithm B solving **FindPrimeFactorBetween**. We can create B using A as a subroutine in a reasonably straightforward way: $B(x, \mathit{lo}, \mathit{hi})$ uses binary search on the interval $[\mathit{lo}, \mathit{hi}]$ and calls to A to zero in on a single prime factor of x in the interval (or else reports at the outset that none exists).

In the other direction, any algorithm for the search problem **FindPrimeFactorBetween** can immediately be used to solve the decision problem **ExistsPrimeFactorBetween**.

Exercise. A mildly sophisticated algorithms problem: Given access to algorithm (subroutine) for solving the decision problem **ExistsPath**, come up with a "simple" algorithm that solves the search problem **FindPath**.

Hint. Given $\langle G, s, t\rangle$, the obvious first step is to call **ExistsPath**($\langle G, s, t\rangle$) to see if there is any path at all. If so, consider trying each neighbor v of s and calling **ExistsPath**($\langle G_{-s}, v, t\rangle$), where G_{-s} denotes the graph G with vertex s (and the edges touching it) deleted...

4 Equivalence of decision problems and languages

As mentioned, we mostly focus on decision problems in Complexity Theory. An *exactly equivalent* concept is that of *languages*. "Languages" and "decision problems" are just two different ways of looking at the same thing, and it is popular in Complexity Theory (somewhat for historical reasons) to focus more on languages.

Definition (Language)**.** A *language* (over alphabet Σ) is any subset of Σ^*; i.e., any set of strings.

Example (Primes)**.** The language **Primes** $\subseteq \Sigma^*$ is all those strings encoding a prime number. That is,

$$\textbf{Primes} = \{\langle x\rangle : x \in \mathbb{N} \text{ is prime}\} = \{\texttt{10}, \texttt{11}, \texttt{101}, \texttt{111}, \texttt{1011}, \dots\}.$$

The language **Primes** corresponds to the decision problem **IsPrime** from Example (IsPrime). More generally, the correspondence between decision problems and languages is as follows:

- Decision problem $f : \Sigma^* \to \{\text{no}, \text{yes}\}$ corresponds to the language $L = \{w \in \Sigma^* : f(w) = \text{yes}\}$.
- Conversely, language $L \subseteq \Sigma^*$ corresponds to the decision problem $f : \Sigma^* \to \{\text{no}, \text{yes}\}$ defined by

$$f(w) = \begin{cases} \text{yes} & \text{if } w \in L \\ \text{no} & \text{if } w \notin L. \end{cases}$$

Exercise. What is the language (over $\{\texttt{0}, \texttt{1}\}$) corresponding to the decision problem **ExistsPath** from Example (ExistsPath)?

Solution. **Path** $\subseteq \{\texttt{0}, \texttt{1}\}^*$ defined by

$$\textbf{Path} = \{\langle G, s, t\rangle : \text{ there is a path from vertex } s \text{ to vertex } t \text{ in graph } G\}.$$

■