Supplementary Chapter of *Theoretical Computer Science for the Working Category Theorist*

Noson S. Yanofsky[1]

October 27, 2019

[1]Department of Computer and Information Science, Brooklyn College CUNY, Brooklyn, N.Y. 11210. And Computer Science Department, The Graduate Center CUNY, New York, N.Y. 10016. e-mail: noson@sci.brooklyn.cuny.edu

Contents

Chapter 7

Other Fields of Theoretical Computer Science

While computability theory and complexity theory are the central parts of theoretical computer science, it is only the beginning. There are many other parts of this fascinating, ever-expanding field.

7.1 Formal Language Theory

The limits of my language means the limits of my world.
Ludwig Wittgenstein
Tractatus 5.6

The idea behind formal language theory is that there is an intimate relationship between the complexity of a machine and the complexity of the language it understands. The more sophisticated the machine, the more sophisticated is the language it understands. One sees this in everyday life. For example, a child's language is not as sophisticated as an adult's language because the child lacks the learning and the experience of the adult. A more educated adult will understand a more sophisticated language than a less educated adult.

The Church-Turing thesis has taught us that all of our usual computational models are basically equivalent. Where are we going to find machines of different levels of sophistication? We have to look at machines that are *weaker* than Turing machines, register machines, and Boolean circuits. A simple example of a weaker machine is a soda machine. Such a machine is not very sophisticated. It understands the language of $1.35. The machine

takes nickels, dimes, and quarters till you reach 135 cents. Once the soda machine sees that you have the right amount, it will shower you with its blessings. In this section we examine various machines and their relationship to associated languages.

First, some preliminaries about languages.

- An **alphabet** is a finite set of symbols denoted Σ. Examples are $\Sigma = \{a, b, c, \ldots, z\}$, $\Sigma = \{a, b\}$, or a favorite of computer science: $\Sigma = \{0, 1\}$.

- For every Σ, a **word** is a sequence of symbols from the alphabet. For Σ = {a, b, c, …, z}, some words are "cat" and "balderdash." Words for Σ = {a, b} are "bbbbab" and "ababab". The set of words made of two symbols is denoted Σ^2. Words of n symbols are denoted Σ^n. The set of all words is denoted Σ^*. While Σ is finite, Σ^* is countably infinite.

- A **language** is a subset of all words. For example with Σ = {a, b, c, …, z} we have English$\subset \Sigma^*$. French, Italian, and Spanish are other languages for the same alphabet. For the alphabet Σ = {0,1,2, …, 9} we will look at the language $\{w | w$ is a prime number$\}$. For $\Sigma = \{a, b\}$ we will look at languages like $\{a^m b^n | m > 0, n > 0\}$ and compare them to $\{a^n b^n | n > 0\}$. A language is either finite or countably infinite.

- The set of **all languages** is the set of all subsets of words. In symbols this is $\mathscr{P}(\Sigma^*)$ where $\mathscr{P}$ is the powerset function. There are uncountably infinite many languages.

- We will mostly be interested in describing a **class of languages** which is a subset of all languages. $C \subseteq \mathscr{P}(\Sigma^*)$. We will be looking at various classes of languages that are described by computational devices of different power.

Understanding a language means determining which words are in the language and which words are not.

Example 7.1.1. Some simple examples are needed. Let us think of numbers as strings of digits.

- A typical first grader can determine the language $\{n \in \mathscr{N} : n$ is even$\}$. This means that if we gave a number to a first grader, the child will be able to say if the number is even or not.

- A slightly more complicated language that a fourth grader can determine is the language $\{n \in \mathscr{N} : n$ is prime$\}$.

- One can go to the extreme and ask about the language

$$\{c \in \mathscr{C} | \text{ there exists a } c' \text{ such that } |c'|^2 > |c|^2 \text{ and } c'$$

$$\text{violates the Riemann conjecture}\}.$$

  As the Riemann conjecture has not been proven or disproven, no one can decide if any complex number is in this language.

□

Since we deal with determining if a word is in a language or not, we will only talk about machines that solve decision problems. The machines will not have output tapes. After reading their input, the machines will either accept, reject or go into an infinite loop (i.e., answer "Yes,""No", or"Maybe".) For every machine M that computes a decision problem, there is a corresponding language

$$L(M) = \{w \in \Sigma : \text{ machine } M \text{ accepts word } w\} \tag{7.1}$$

(i.e., a set of words that are accepted by that machine.) This is like a characteristic function that goes from the set of all words to $Bool$. The characteristic function describes those words that are in the language.

Researchers have described a four-level hierarchy of machines and their languages that is called the **Chomsky hierarchy**. There is nothing complete about this listing. There are many other types of machines and languages classes that can be described. However, convention dictates that we describe these four types of machines and the languages they recognize. First let us meet the machines. They are essentially Turing machines and Turing machines with restrictions on how their input tapes and work tapes are used.

- **Turing machines**. We are already familiar with these machines and we know that they can preform any task that a modern computer can perform. They have unlimited amount of work space on the work tapes. Notice also that they have the ability to enter an infinite loop.

- **Linear bounded automata**. These are like Turing machines but they do not have unlimited work space. A linear bounded automaton does not use more space on the work tape than the size of its input.

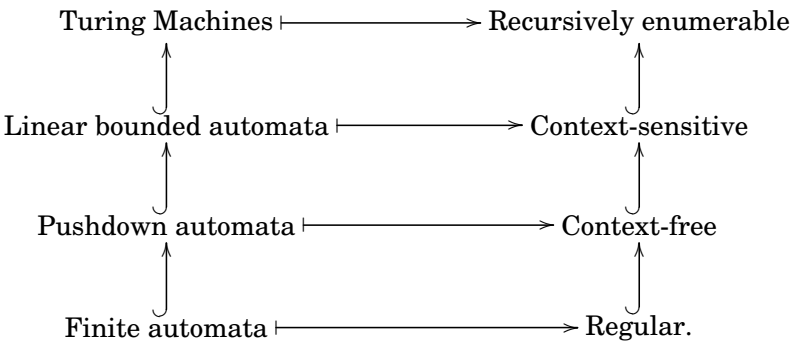
- **Pushdown automata**. These are nondeterministic Turing machines where there is a single work tape which can be used as a primitive type of memory called a "stack." A stack is like a pile of plates in a restaurant. Each memory place is like a plate. When you add a plate, you "push" it onto the top of the stack. One can"pop" off an element from

the memory by removing the top plate of the stack. We say that this memory is "LIFO" which stands for "Last In, First Out". The memory is not random access. With a pushdown automata, one can not access all the memory. There is also a restriction that the pushdown automata read its input from left to right without reviewing.

- **Finite automata**. This is the simplest type of machine we will discuss. In such machines, there are no work tapes and the input tapes are read from left to right without the ability to write on them. This means there is essentially no memory. The computer just reads the input once from left to right. The machine can change states for every letter. When it completes the input, if the final state is an accepting state, the word is accepted. Otherwise it is rejected.

These machines are included in each other and the L function takes a class of machines to a class of languages as follows

$$\textbf{Machines} \longrightarrow \textbf{Languages} \tag{7.2}$$



What are these language classes? Computer scientists describe languages with **grammars**. These different languages can be described by different types of grammars. Because of space restrictions we will not describe these types of grammars. Rather, we will list off the language classes and describe an instructive example of a languages in that class of languages. We will start from the smallest language class.

- **Regular languages**. A typical language in this class is

$$\{a^m b^n | m,n \in \mathcal{N}\}.$$

In order to determine if a word is in this language, a Turing machine must read the data and make sure all the a's come before all the b's.

Notice that the machine does not have to count how many a's or b's there are. No memory is required.

- **Context-free languages**. A typical language in this class is

$$\{a^m b^m | m \in \mathcal{N}\}.$$

In order to determine if a word is in this language, a Turing machine must read the data and make sure all the a's come before all the b's. At the same time, the machine must insure that the number of a's is the same as the number of b's. This can be done by adding an element to the stack every time one sees an a and removing an element from the stack every time one sees a b. The word will be accepted if the word ends with an empty stack. Notice that this language cannot be decided by a finite automata because it demands some work tape.

- **Context-sensitive languages**. A typical language in this class is

$$\{a^n b^n c^n | n \in \mathcal{N}\}.$$

There is no way that this language can be recognized with one stack. One demands at least two stacks. Another way to decide this language is to make sure the input is of the correct form and then, for each a make sure there is a unique b and a unique c. This can be done with a linear bounded automata but not a push-down automata.

- **Recursively enumerable language**. An example of a language that is recursively enumerable is

$$\{a^x b^y \quad | \quad \text{Turing machine } y \text{ on input } x \text{ halts}\}.$$

To determine if a word is in this language, a Turing machine would have to count how many a's and b's there are. The Turing machine would then have to simulate Turing machine y on input x and see if it halts. If it does halt, then the Turing machine will know that the word is in the language. However, Turing machine y on input x can enter an infinite loop and the computer will not be able to reject the word.

Many textbooks in theoretical computer science start with the definition of a finite automaton and then "build up" to more powerful computers like Turing machines. Since they start with this basic concept, there are many

well-known ideas about finite automata. We would be remiss to have a textbook on theoretical computer science and not spend some time on these machines.

As we saw above, a finite automaton is a simple machine. There is a beautiful categorical way of viewing finite automata that we shamelessly adopt from Section 5.3 of [25]. We begin by considering the category of finite graphs and graph homomorphisms which is defined as

$$\mathbb{F}\mathrm{in}\mathbb{G}\mathrm{raph} = \mathbb{F}\mathrm{in}\mathbb{S}\mathrm{et}^{\,*\rightrightarrows *}. \tag{7.3}$$

By finite graphs we mean that both the vertices (which will correspond to the states) and the arrows (which will correspond to the alphabet) are finite. There are many examples of finite graphs. Let us consider two of them.

Example 7.1.2. Let $\mathbf{2_0}$ be the graph with two objects and no arrows. We will call the two objects s and t for reasons that will become obvious.
$\mathbf{2_0} = s \qquad t$. □

Example 7.1.3. For each alphabet Σ there will be a graph Σ with only one object. For every symbol in the alphabet, there will be one arrow from the single object to itself. There will be one extra arrow that will be labeled with the Greek letter epsilon ϵ. Such graphs might look like these

(7.4)

□

Consider the slice category $\mathbf{2_0}/\mathbb{F}\mathrm{in}\mathbb{G}\mathrm{raph}$. This is the category of finite graphs with distinguished vertices which we will call s and t. (We have no problem if $s = t$.) We call such graphs "doubly-pointed graphs." A graph homomorphism between doubly-pointed graphs must take s to s and t to t. For every alphabet Σ, the graph Σ is also a doubly-pointed graph.

Finally we come to the definition of the category of finite automata.

Definition 7.1.4. For any alphabet Σ we define the **category of finite automata** with symbols in Σ as

$$\mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma = (\mathbf{2_0}/\mathbb{F}\mathrm{in}\mathbb{G}\mathrm{raph})/\Sigma. \tag{7.5}$$

Let us spell this out. The objects will be finite doubly pointed graphs and every arrow will be labeled with a letter of Σ or ϵ. The morphisms will be

graph homomorphisms that take s to s and t to t. The graph homomorphisms are also required to take every arrow in one graph to an arrow with the same label. We will describe a finite automaton as $M = s \rightsquigarrow G \rightsquigarrow t$. ◇

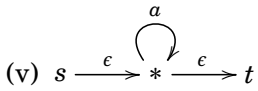
Example 7.1.5. Here are some finite automata (we will show what languages they describe in Example 7.1.6.)

(i) $s \qquad t$. (ii) $s \xrightarrow{a} t$ where a is a letter in the alphabet.

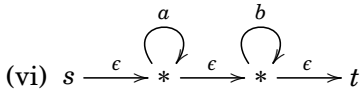
(iii) $s \xrightarrow{a} * \xrightarrow{b} t$ where a and b are letters in the alphabet.

(iv) $s \xrightarrow{a} * \xrightarrow{\epsilon} * \xrightarrow{b} t$ where a and b are in the alphabet.

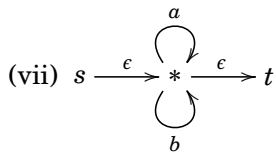
(v)



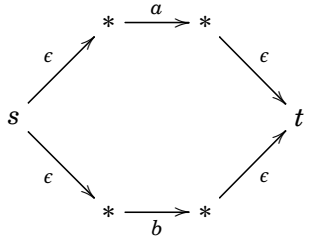
(vi)



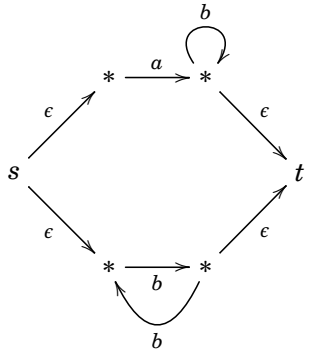
(vii)



(viii)



(ix)



□

A finite automaton describes the words in a regular language. The words are the paths from s to t in the finite automaton. If at any vertex there is an arrow with an ϵ then the path can cross that arrow for free. Categorically, we can describe the set of words of a Turing machine by using the adjunction between small categories and graphs. For every finite graph G there is a free category $Free(G)$ which has the same objects as G. The arrows are finite composable sequences of arrows of G. The $Free$ functor can be extended to the slice and coslice categories and hence to $\mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma$. The language of the finite automaton M is

$$L(M) = Hom_{Free(M)}(s,t). \tag{7.6}$$

This, in fact, describes a functor $L : \mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma \longrightarrow \mathbb{R}\mathrm{egular}$.

Example 7.1.6. Let us determine what languages are described by the finite

automata of Example 7.1.5.

(i) The empty language. There are no words in this language.

(ii) This finite automaton describes the language that consists of one word a. That is, the language $\{a\}$. (iii) The language $\{ab\}$.

(iv) The language $\{ab\}$. The point here is that the ϵ does not make a difference. (v) $\{a^m : m \in \mathcal{N}\}$.

(vi) $\{a^m b^n : m,n \in \mathcal{N}\}$. (vii) The language is any word. That is $\{a,b\}^*$.

(viii) $\{a\} \cup \{b\} = \{a,b\}$. (ix) $\{ab^m : m \in \mathcal{N}\} \cup \{b^{2t+1} : t \in \mathcal{N}\}$. □

There are three operations that one can perform on the objects in the category $\mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma$.

- **Concatenation operation**. Given finite automata

$$s \rightsquigarrow G \rightsquigarrow t \quad \text{and} \quad s' \rightsquigarrow G' \rightsquigarrow t' \ ,$$

we can construct the finite automaton

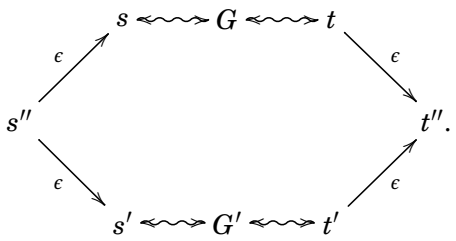
$$s \rightsquigarrow G \rightsquigarrow t \xrightarrow{\epsilon} s' \rightsquigarrow G' \rightsquigarrow t'. \tag{7.7}$$

If L and L' are the languages that correspond to G and G' respectively, then the language that corresponds to the concatenation of these finite automata corresponds to $L \circ L' = \{w : w = w_1 w_2, w_1 \in L, w_2 \in L'\}$.

- **Union operation**. Given finite automata

$$s \rightsquigarrow G \rightsquigarrow t \quad \text{and} \quad s' \rightsquigarrow G' \rightsquigarrow t' \ ,$$

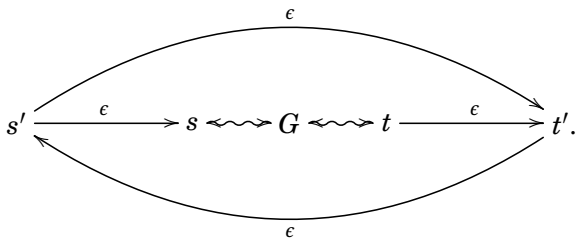
we can construct the finite automaton



(7.8)

If L and L' are the languages that correspond to G and G' respectively, then the language that corresponds to the union of G and G' corresponds to $L \cup L'$.

- **Star Operation**. Given finite automaton $s \rightsquigarrow G \rightsquigarrow t$ we can construct the finite automaton



(7.9)

  If L is the language that corresponds to G, then the language that corresponds to the star of g corresponds to L^* which contains concatenations of words in L.

These operations generate the category of finite automata.

Theorem 7.1.7. Kleene's Theorem. Every object in the category $\mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma$ is equivalent to some finite automaton that is generated by the empty language, or the single letters and the operations of union, concatenation, and the star operation. ★

While this is a nice categorical way to see finite automata, it is not the standard definition. The usual definition is as follows.

Definition 7.1.8. A **deterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, s, F)$ where

- Q is a finite set of states.
- Σ is a finite alphabet.
- $\delta : Q \times \Sigma \longrightarrow Q$ is a transition function that accepts a state and a letter and tells what state to go into. (This is a truncated version of a transition function that we saw in Diagram (**??**) of a deterministic Turing machine. However, here, where there is no writing on tapes and the direction is forced to always be right.)
- $s \in Q$ is the starting state.
- $F \subseteq Q$ is the set of accepting states.

◇

While this definition is fine, one can soup-up the definition to get a seemingly more powerful structure.

Definition 7.1.9. A **nondeterminstic finite automaton with ϵ-moves** is a 5-tuple $(Q, \Sigma, \delta, s_0, F)$ where everything is as in Definition 7.1.8 except

- $\delta : Q \times (\Sigma \cup \epsilon) \longrightarrow \mathscr{P}(Q)$ is a transition function that accepts a state and a letter or an ϵ and tells what set of possible states to go into. (This is a truncated version of a transition function that we saw in Diagram (**??**) of a nondeterministic Turing machine. However, here, where there is no writing on tapes and the direction is forced to always be right.)

◇

Exercise 7.1.10. Show that every nondeterminstic finite automaton with ϵ-moves has an equivalent nondeterminstic finite automaton and ϵ-moves with a single accepting state.

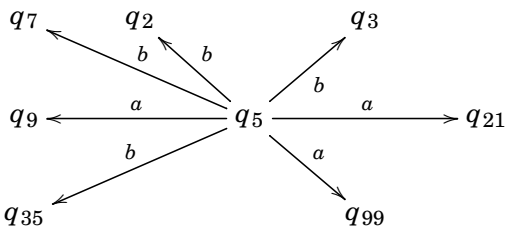
Solution: Add one new state, i.e., $Q' = Q \cup \{t\}$ and make an ϵ-move from each of the states in F to t. This ensures that any time the machine is in some accepting state, it is automatically in t. ■

Exercise 7.1.11. Show that every nondeterminstic finite automaton with ϵ-moves as defined in Definition 7.1.9 is equivalent to an object of $\mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma$ as defined in Definition 7.1.4.

Solution: The states Q are the objects of the finite graph. The δ describes the arrows from state to state labeled by letters of the alphabet. s is the starting state. The last Exercise shows that one needs a single accepting state. ■

The relationship between the category of deterministic finite automata $\mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma$ and the category of nondeterministic finite automata with ϵ-moves $\mathbb{NF}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma$ is worthy of thought. Every deterministic finite automaton is a type of nondeterministic finite-automaton where there is exactly one state that one can go to and there is no ϵ move. This means that there is an inclusion functor $Inc : \mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma \hookrightarrow \mathbb{NF}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma$

On the other hand, for every nondeterministic finite automata with ϵ-moves, there exists a deterministic finite automata that accepts the same language. The idea behind this construction is that for a nondeterministic finite automaton M we can construct deterministic finite automaton $F(M)$ that recognizes the same language as M. The set of states for $F(M)$ will be the powerset of the set of states of M. In other words, if Q is the set of states of M, then $\mathscr{P}(Q)$ is the set of states of $F(M)$. The transitions for $F(M)$ are best described with an example. If the transitions for M out of one state looks like this:



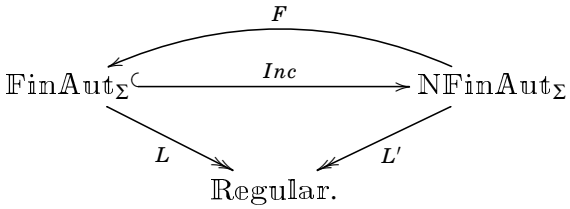
(7.10)

then the transactions for $F(M)$ out of that same state looks like this

$$\{q_9, q_{21}, q_{99}\} \xleftarrow{a} \{q_5\} \xrightarrow{b} \{q_{35}, q_7, q_2, q_3\}. \tag{7.11}$$

This is a deterministic finite automaton that has the same information as the original nondeterministic finite automata.

We have just essentially defined a functor $F\colon \mathbb{NFinAut}_\Sigma \longrightarrow \mathbb{FinAut}_\Sigma$. Inc and F are related as follows:



(7.12)

The two triangles in the diagram commute and $F \circ Inc = Id$ but $Inc \circ F$ need not equal the identity.

Advanced Topic 7.1.12. Given a program, one can ask if it is the shortest program that performs the task. It turns out that for a modern computer (Turing machine) there is no way to determine if the given program is the shortest program or not. In contrast, finite automata are simple enough machines that there is a procedure that can take a finite automaton and find a finite automaton that accepts the same language but has the fewest states. This minimal state finite automata is universal in the categorical sense. See Section 3.4 of [10], Section 9.7 of [7], and Section 5.7 of [19]. ○

Research Project 7.1.13. A nice research project is to extend the categorical definition of the category of finite automata given in Definition 7.1.4 to formulate the notion of the category of pushdown automaton. This demands that every edge of a graph no only get a letter to read but also a letter to potentially push onto a stack. Another letter is needed to decide what to do when one pops an element off a stack. While you are at it, you might as

well extend this extension to formulate the notion of a pushdown automaton with two stacks. It turns out that any pushdown automaton with two stacks is equivalent to a Turing machine. See if you make an equivalence between the category of two-stacked pushdown automatas and the the category $\mathbb{T}\mathrm{uring}(1,1)$.

Another nice small project is to formulate the union, concatenation and star operations as functors on the category of finite automata and as functors of regular languages. Make sure the operations as functors agree with the functor L from the category of finite automata to the category of regular languages. This means that the following square should be commutative.

$$\begin{array}{ccc} \mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma \times \mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma & \xrightarrow{Operation} & \mathbb{F}\mathrm{in}\mathbb{A}\mathrm{ut}_\Sigma \\ \downarrow{\scriptstyle L\times L} & & \downarrow{\scriptstyle L} \\ \mathbb{R}\mathrm{egular} \times \mathbb{R}\mathrm{egular} & \xrightarrow[Operation]{} & \mathbb{R}\mathrm{egular}. \end{array} \tag{7.13}$$

♣

Further Reading

Chapters 1-4 of [22], Chapters 7-9 of [7], Chapters 5-16 of [19] and Chapters 2-6 of [10]. Our categorical presentation of finite automata gained much from Section 5.3 of [25]. There is an entire book about finite automata from a categorical perspective that is interesting: [8].

7.2 Cryptography

Few persons can be made to believe that it is not quite an easy thing to invent a method of secret writing which shall baffle investigation. Yet it may be roundly asserted that human ingenuity cannot concoct a cipher which human ingenuity cannot resolve.

Edgar Allan Poe
"A Few Words on Secret Writing" in Graham's Magazine (July 1841).

Traditionally cryptography is the science of communicating hidden messages in the presence of adversaries. Modern cryptography uses ideas from complexity theory to make sure that any adversary that intercepts the secret messages will have a hard time decoding the messages. That is, it will be computationally *in*efficient for the adversaries to decode the messages.

Before we go further, we need to learn some of the nomenclature of cryptography. We usually have "Alice" trying to communicate something to "Bob." Alice's message starts in "plaintext" and she uses a "cipher" to encode the plaintext into "ciphertext." The ciphertext is sent to Bob who decodes it back to plaintext. There is also an eavesdropper named "Eve" who can intercept the ciphertext. In our use, the plaintext will be described by a sequence of data types called $SeqA$ and the ciphertext will be a sequence of types called $SeqB$. Alice will use a computable function $e\colon SeqA \longrightarrow SeqB$ to encode plaintext to ciphertext. Bob will use a computable function $d\colon SeqB \longrightarrow SeqA$ to decode the ciphertext into plaintext. Eve will see the ciphertext and try to find the correct decoder. Hopefully she will have a hard time doing this.

The power of category theory will be obvious here. We will describe a simple categorical structure and every major cryptographic protocol will be an instance of this categorical structure. In order to do this we will need to define $\mathbb{E}\mathrm{asy}$ and **Hard** which will correspond to functions that are easy to compute and hard to compute. $\mathbb{E}\mathrm{asy}$ will be a subcategory of $\mathbb{T}\mathrm{ot}\mathbb{C}\mathrm{omp}\mathbb{F}\mathrm{unc}$ which contains all the identities of $\mathbb{T}\mathrm{ot}\mathbb{C}\mathrm{omp}\mathbb{F}\mathrm{unc}$ and has morphisms that do not demand much computational resources. In contrast, **Hard** will not be a subcategory but a subset of morphisms in $\mathbb{T}\mathrm{ot}\mathbb{C}\mathrm{omp}\mathbb{F}\mathrm{unc}$ that are not in $\mathbb{E}\mathrm{asy}$.

We are being intentionally vague about the morphisms in $\mathbb{E}\mathrm{asy}$ and **Hard**. Here are some of the many possibilities.

- The obvious choice is to have $\mathbb{E}\mathrm{asy} = \mathbb{DTIME}(Poly)$, that is all polynomial time deterministic functions and let **Hard** be all functions that use exponential amount of time or worse. The problem with this, is

that a function that runs in n^{100} is polynomial but is not exactly easy to compute.

- $\mathbb{E}\mathrm{asy}$ can be any function under n^5 and leave **Hard** as functions that demand an exponential amount of time.

- Researchers are already talking about "post-quantum cryptography." This is where we assume that Eve has a large-scale quantum computer and will use it on the ciphertext. In that case, $\mathbb{E}\mathrm{asy}$ will be functions that demand time in the low polynomials but **Hard** should be functions that demand more resources than exponential time. Perhaps exponential space.

With $\mathbb{E}\mathrm{asy}$ and **Hard** in hand, we make the following simple definition as a nice steppingstone for our ultimate goal of a single structure which describes many cryptography protocols. The intuition is that there is an easy encoding function that is hard to decode.

Definition 7.2.1. A **one-way function** is a morphism $e: SeqA \longrightarrow SeqB$ in $\mathbb{E}\mathrm{asy}$ such that any morphism $d: SeqB \longrightarrow SeqA$ in $\mathbb{T}\mathrm{ot}\mathbb{C}\mathrm{omp}\mathbb{F}\mathrm{unc}$ with the property $d \circ e = Id_{SeqA}$ is in **Hard**. ◇

Example 7.2.2. The simplest example of a one-way function is multiplication. The computable function $Mult: Nat \times Nat \longrightarrow Nat$ is defined as $Mult(m,n) = m \cdot n$ if m and n are both primes (set the result to 0 if either number is not a prime). This function is polynomial. In contrast, an inverse of this function would have to factor the number. At this time, we can factor an x with a search through many numbers less than $\sqrt{x}$ to find a factor. This can be an exponential function in terms of the size of x. □

Example 7.2.3. Another example of a one-way function is modular exponentiation. As we saw in Example **??**, in contrast to real numbers where one can easily invert the exponentiation function by taking the logarithm of the number, for the discrete case, there is no known easy way of inverting the exponentiation function. The function $f: Nat \times Nat \times Nat \longrightarrow Nat \times Nat \times Nat$ defined as

$$f(n,b,x) = (n,b,b^x Mod\ n) \tag{7.14}$$

is polynomial in the size of the input. In contrast, given $(n,b,b^x Mod\ n)$ one has to search through many possible values to find x. This might be exponentially time consuming and is called the **Discrete logarithm problem**. □

While the definition of a one-way function is interesting and the two examples given are worthy of deeper thought, it is not very applicable. One of the main tenets of cryptography is that the intended receiver of the secret message should be able to easily decode the message. In other words, the function $d: SeqB \longrightarrow SeqA$ should also be a morphism in $\mathbb{E}\mathrm{asy}$. It should not be hard to decode, rather, it should be hard *to find the right decoder*. The notion of a trapdoor function will be helpful. While it is hard to find the right decoder, with the right key, it will be easy to find the right decoder. A trapdoor is a secret door in the floor that one can easily fall into, but is hard to get back out. (Some of the literature gives the definition of a trapdoor function in terms of the probability of finding the correct decoder. Our definition is based on the complexity of finding the correct decoder.)

Definition 7.2.4. A **cryptographic protocol** that encodes data of type $SeqA$ into data of type $SeqB$ consists of

(i) a set of "encoder" functions, $Enc \subseteq Hom_{\mathbb{E}\mathrm{asy}}(SeqA, SeqB)$,

(ii) a set of "decoder" functions, $Dec \subseteq Hom_{\mathbb{E}\mathrm{asy}}(SeqB, SeqA)$,

(iii) an "inverter" function $INV: Enc \longrightarrow Dec$ in **Hard** such that for all $e \in Enc$ there is a $d = INV(e)$ that satisfies $d \circ e = Id_{SeqA}$,

(iv) a "key" function $KEY: Enc \longrightarrow Seq$ in **Hard** such that for all $e \in Enc$ there is a $k_e = KEY(e)$, and

(v) a "trapdoor" function $TRP: Seq \longrightarrow Dec$ in $\mathbb{E}\mathrm{asy}$ satisfying

$$\begin{array}{ccc} & Seq & \\ {\scriptstyle KEY}\nearrow & & \searrow{\scriptstyle TRP} \\ Enc & \xrightarrow[INV]{} & Dec \end{array} \tag{7.15}$$

i.e., for every $e \in Enc$ there is a "key" $k_e \in Seq$ such that $TRP(k_e) = INV(e)$.

◇

First we will describe a few simple protocols to get our feet wet.

Example 7.2.5. Probably the simplest cryptographic protocol is the **Caesar's cipher**. This supposedly goes back to a method used by Julius Caesar to communicate with his generals. He changed the plaintext by exchanging every letter in the alphabet with another letter shifted over a fixed amount.

If you shift the letters in the English alphabet 6 positions then it looks like this:

$$A \mapsto G, B \mapsto H, \ldots, T \mapsto Z, U \mapsto A, V \mapsto B, W \mapsto C, X \mapsto D, Y \mapsto E, Z \mapsto F. \tag{7.16}$$

A secret message would then change from "ATTACK AT DAWN!" to "GZZGIQ GZ JGCT!"

In terms of our definition, the Caesar cipher is a cryptographic protocol from $String$ to $String$. (i) There are 26 functions in $Enc = \{f_1, f_2, f_3, \ldots f_{26}\}$. Each one, simply shifts over the letters a fixed amount. (ii) $Dec = Enc$. (iii) For each $f \in Enc$, there 26 possible values for $INV(f)$. This is not so bad in terms of modern computers. Even before computers came along, the Caesar cipher was not a very effective cryptographic protocol. (iv) For every map $f_x \in Enc$, the key is the number x. (v) The trapdoor function $TRP : Nat \longrightarrow String^{String}$ is defined as $TRP(x) = f_{-xMod26}$. Its not hard to see that $TRP(x) \circ f_x = Id$. □

Example 7.2.6. A slightly more complicated cipher is called a **Substitution cipher**. Alice encodes her message to Bob by exchanging some letter for some other letter. That is, there is some permutation of the letters. There are 26! such permutations. (i) Enc will consist of 26! maps. Each permutation π of the letters will determine a function f_π that changes a plaintext message into a ciphertext message by swapping the letters. (ii) $Dec = Enc$. (iii) Theoretically, for Eve to find the value of $INV(f)$ she would have to go through all possible decoding functions in Dec to recover the plaintext. (How would she know when she has the right plaintext?) (iv) The key for f_π is π. (v) The trapdoor function inputs the permutation used and outputs the inverse permutation function, i.e., $TRP(\pi) = f_{\pi^{-1}}$.

Since 26! is an astoundingly large number, the function INV is in anyone's definition of **Hard** and this protocol seems secure. However, it is not! This code can easily be broken by statistical analysis. Every language has certain statistical facts that make this protocol breakable. For example, in English text, the letters "e," "t," and "a" are the most popular letters. In contrast, the letters "j," "q," and "z" are the least popular letters. Every "q" is followed by "u." Many words start with "th," etc. In order to break the code, perform a statistical analysis on the ciphertext and make some educated guesses as to what letters are substituted. Eve has it pretty easy. □

Example 7.2.7. A **one-time pad cipher** uses a random text to form the ciphertext. The name comes from the fact that the random text can only be used once. If it is used more than once, the cipher might be broken. For

our purposes, this cipher encodes $Bool^*$ to $Bool^*$. The random text is also a string of $Bool$ and must be at least the same size as the text. Alice and Bob must agree on the one-time pad beforehand. Given a plaintext, the cipher does an exclusive or (XOR) on every bit of the plaintext with the one-time pad to get the ciphertext as follows.

	0	1	1	1	0	1	0	plaintext
$\oplus$	1	1	0	1	0	0	1	one-time-pad
	1	0	1	0	0	1	1	ciphertext

To decode, Bob simply has to do the same operation to the ciphertext. To see this, think of p as a bit of plaintext, t as a bit of the one-time pad and c as a bit of the ciphertext. Using this, we can describe the cipher as $p \oplus t = c$. To decode, XOR with the one-time pad to get $c \oplus t = p \oplus t \oplus t = p \oplus 0 = p$. The reason why it is called a one-time pad is because if you use the same one-time pad for both p and p' then you will get $p \oplus t = c$ and $p' \oplus t = c'$. Eve can then take c and c' and XOR them to get $c \oplus c' = p \oplus t \otimes p' \oplus t = p \oplus p' \oplus t \oplus t = p \oplus p'$. From here, it is not hard to extract p and p'.

Let us describe this protocol using our categorical structure. (i) For a plaintext message of size n, there will be 2^n one-time-pads and hence Enc has 2^n encoders. (ii) Once again, $Dec = Enc$. (iii) Given the ciphertext, only a brute-force search through all one-time-pads will give you the inverse function. Since there are 2^n such one-time pads, this is exponentially hard. (iv) The key is the one time pad. (v) The trapdoor function takes the one-time pad P and outputs the function that uses that pad, i.e., $TRP(P) = f_P$.

The one-time pad is essentially secure unless Alice uses it twice. The problem is that there needs to be a lot of shared one-time pads. This is not always easy. This brings us to our next idea. □

So far, all the examples that we have seen are called **private key protocols** or **symmetric key cryptography**. This means that the key used to decode is essentially the same as the code used to encode. The keys shared by Alice and Bob are symmetric and must be kept private. This brings to light many other issues in cryptography. There is "key generation" (how are all these keys randomly created,) "key establishment" (how are the keys distributed or agreed on,) and "key management" (how are the keys stored and kept secret.) Also, notice that if Bob wants to communicate with Ann, Angelina, and Aishwarya, he has to have shared keys with each of them. Confusing the keys could get embarrassing.

Cryptography went through a major revolution at the end of the 1970s with the advent of **public key protocols** and **asymmetric key cryptography**. This is where Bob publicly announces a public key for anyone to communicate with him. The decoding key will be different (or asymmetric) from the encoding key. Much internet communication is performed with these ideas.

The paradigm example of public key protocols is RSA.

Example 7.2.8. The **RSA protocol** was invented by Ron Rivest, Adi Shamir, and Leonard Adleman, in 1978 (but there is an interesting prehistory that is worthy of looking into.)

Before anyone can communicate with Bob, he must perform the following preliminary tasks.

(I) Randomly choose two large prime numbers p and q.

(II) Multiply these number to get $n = p \cdot q$.

(III) Determine how many numbers less than n are relatively prime to n. (That is, the number of elements less than n that do not have any non-trivial common factor with n, i.e., determine the number of $x < n$ such that $GCD(n,x) = 1$.) The function this describes is called the **Euler's totient function** and is denoted $\phi(n)$. Since p and q are both prime we have $\phi(p) = p-1$, $\phi(q) = q-1$ and $\phi(n) = \phi(p \cdot q) = (p-1)(q-1)$.

(IV) Randomly choose a number $e < n$ that is a relatively prime to $\phi(n)$.

(V) Calculate d such that

$$e \cdot d \equiv 1 \; Mod \; \phi(n) \quad \text{or in other words} \quad e \cdot d = 1 + k\phi(n) \text{ for some } k.$$

All five of these steps can be done with ease in polynomial time. Bob makes (e,n) to be the public key and he keeps (d,n) as his private key.

We need to consider functions of the form $f_{g,x}$ which are defined as $f_{g,x}(m) = m^g \; Mod \; x$. This operation is called **modular exponentiation.** There are nice computational tricks to perform these operations in polynomial time or less.

Let us formally go through the RSA cryptographic protocol that encodes Nat into Nat.

(i) Since Bob told us how anyone can communicate with him, we can set $Enc = \{f_{e,n}\}$ to be the singleton set.

(ii) In contrast, there are many possibilities for decoders. Basically the decoder can use any exponent less n that could factor $\phi(n)$

$$Dec = \{f_{x,n} : x < n\}. \tag{7.17}$$

(iii) One possible way of finding the right $INV(f_{e,n})$ is to factor n into p and q. With p and q it is easy to determine $\phi(n)$ and then d. However to determine these factors is exponentially hard in the size of n.

(iv) $KEY(f_{e,n}) = d$.

(v) Bob can easily decode the ciphertext by using the trapdoor function $TRP(d) = f_{d,n}$.

Why does RSA work? Let us say that Alice wants to communicate the number m to Bob. Alice uses the public key to get

$$f_{e,n}(m) = m^e \ Mod \ n = c. \tag{7.18}$$

She then sends c to Bob. He uses $f_{d,n}$ on c to get

$$f_{d,n}(c) \equiv c^d \ Mod \ n \equiv (m^e \ Mod \ n)^d \ Mod \ n \equiv (m^e)^d \ Mod \ n \equiv m^{c \cdot d} \ Mod \ n. \tag{7.19}$$

By Equation ((V)), this is equal to

$$m^{1+k\phi(n)} \ Mod \ n \equiv m \times m^{k\phi(n)} \ Mod \ n \equiv m \times (m^{\phi(n)})^k \ Mod \ n. \tag{7.20}$$

Euler's theorem in number theory says

$$m^{\phi(n)} \ Mod \ n \equiv 1 \ Mod \ n \tag{7.21}$$

which means

$$m \times (m^{\phi(n)})^k \ Mod \ n \equiv m \times 1 \ Mod \ n = m \tag{7.22}$$

thus showing that the decryption function works.

It is worth noting that if Eve can factor $n = pq$ then she would be able to find d and decipher any message. While there are no known deterministic polynomial time algorithm to factor numbers, there is a polynomial algorithm [20] on a quantum computer that can factor numbers. As scientists get better and better at making large-scale quantum computers, the usefulness of RSA is less assured. See [30] for an easy introduction to Shor's algorithm for factoring and for quantum computing in general. □

Example 7.2.9. Before we move on to some more protocols, let us take a small interlude and explore another feature of public key cryptography. Let's say that Bob wants to send a message and he wants to make sure that Alice knows the message comes from him and no one else. We assume that $INV(e) = d$ not only satisfies $d \circ e = Id$ but also $e \circ d = Id$. Bob can send a **digital signature** by sending his message m along with $d(m)$. Alice will then take $d(m)$ (which she will not be able to understand) and apply e to it. If $e(d(m)) = m$ then she knows that this message could have only come from Bob. No one else would have the private key d. □

Example 7.2.10. As we saw, one of the hard parts of cryptography is making sure that Alice and Bob have matching keys. The **Diffie-Hellman key exchange protocol** accomplishes this in a clever way. Rather than using the hardness of factoring, this protocol uses the hardness of discrete logarithms which we met in Examples **??** and 7.2.3. The protocol is a five-step process.

(I) Alice and Bob publicly agree on a large prime number p and an integer $\alpha \in \{2,3,\ldots,p-2\}$. (They can do this in public because Eve will not gain by knowing this information.)

(II) Alice randomly chooses a private $a \in \{2,3,\ldots,p-2\}$, calculates $A = \alpha^a \ Mod \ p$ and sends A to Bob. (Eve will not be able to easily compute a.)

(III) Bob randomly chooses a private $b \in \{2,3,\ldots,p-2\}$, calculates $B = \alpha^b \ Mod \ p$ and sends B to Alice. (Eve will not be able to easily compute b.)

(IV) Alice raises B to her secret a number to get $B^a \ Mod \ p \equiv (\alpha^b)^a \ Mod \ p$.

(V) Bob raises A to his secret b number to get $A^b \ Mod \ p \equiv (\alpha^a)^b \ Mod \ p$.

Since $(\alpha^a)^b \ Mod \ p \equiv (\alpha^b)^a \ Mod \ p$, Alice and Bob now share a secret number. Neither Bob nor anyone else knows Alice's a. Neither Alice nor anyone else knows Bob's b. This all comes from the fact that modular exponentiation is in $\mathbb{Easy}$ while discrete logarithm is in **Hard**. □

Example 7.2.11. The **El-gamal encryption protocol** uses the Diffie-Hellman key exchange protocol to communicate. Imagine Alice wants to send the number m to Bob. They simply exchange a secret key as in the previous example. Once this is done, the protocol continues as follows:

(VI) Alice multiplies her secret number by m. That is she calculates $((\alpha^a)^b) \cdot m \ Mod \ p$ and sends it to Bob.

(VII) Bob calculates the inverse of his (their) number $((a^a)^b)^{-1}\ Mod\ p$.

(VIII) Bob deciphers Alice's message by multiplying the inverse with what he receives from Alice.

$$((a^a)^b)^{-1}((a^a)^b)\cdot m\ Mod\ p \equiv 1\cdot m\ Mod\ p = m\ Mod\ p. \tag{7.23}$$

□

Example 7.2.12. Let us simply mention that there are public key cryptography protocols **Elliptic curve encryption protocols** (ECC) that are based on a shared mathematical structures called "elliptic curves." There is an ECC Diffie-Helman protocol, an ECC El-Gamal protocol, etc. Although these systems are more mathematically sophisticated, they have many good features. □

Before we close our discussion of cryptography we should note that there seems to be a close relationship between cryptography and categorical coherence theory. Remember that coherence theory is a type of higher-dimensional algebra that discusses the implications of higher-order axioms. The classical example of coherence theory is Saunders Mac Lane's monoidal categories (see Section VII.2 of [15]). There is a discussion of different operations one can use to reassociate $((AB)C)D$ to $A(B(CD))$. Coherence theory gives conditions that tell when different operations turn out to be the same. We have a similar situation in cryptography. Alice and Bob are performing different operations on data and the protocols tell us what operations to perform so that the messages will be successfully communicated and decrypted.

- In RSA, the central idea is that $(m^e)^d \equiv m^{(e\cdot d)}\ Mod\ n$ or in terms of commuting diagrams

$$\begin{array}{ccc} Nat\times Nat\times Nat & \xrightarrow{(-)^{(-)}\times id} & Nat\times Nat \\ {\scriptstyle id\times\cdot}\downarrow & & \downarrow{\scriptstyle (-)^{(-)}} \\ Nat\times Nat & \xrightarrow[(-)^{(-)}]{} & Nat. \end{array} \tag{7.24}$$

.

- In Diffie-Hellman, the central idea is $(a^a)^b \equiv (a^b)^a\ Mod\ p$ or in terms

of commuting diagrams

$$\begin{array}{ccccc} Nat \times Nat \times Nat & & \xrightarrow{\quad (-)^{(-)} \times id \quad} & & Nat \times Nat \\ \Big\downarrow{\scriptstyle id \times tw} & & & & \Big\downarrow{\scriptstyle (-)^{(-)}} \\ Nat \times Nat \times Nat & \xrightarrow[(-)^{(-)} \times id]{} & Nat \times Nat & \xrightarrow[(-)^{(-)}]{} & Nat. \end{array} \tag{7.25}$$

.

- In El-gamal, the central idea is $((a^a)^b)^{-1}((a^b)^a) \cdot m \equiv m \; Mod \; p$. One can make a similar coherence diagram corresponding to this equation.
- There are similar axioms in the elliptic curves protocol.

There is work to be done in this direction.

Research Project 7.2.13. A nice research project would be to look at some of the new post-quantum cryptography protocols. See if they can be put into our categorical formulation of a cryptographical protocol.

One should also show why post-quantum cryptography will be important when quantum computers come into existence. Formulate the notion of a quantum Turing machine (see, e.g., Chapter 8 of [30]) and the symmetric monoidal bicategory $\mathbb{QTotTuring}$ of all quantum Turing machines. There will be a functor $\mathbb{QTotTuring} \longrightarrow \mathbb{TotCompFunc}$ that takes every quantum Turing machine to the function it calculates. Go on to formulate a subcategory of $\mathbb{TotCompFunc}$ consisting of quantum easy functions and a subset of quantum hard functions. Show that the classical crptographic protocols will collapse with the existence of a quantum computer but that the post-quantum protocols will be successful. ♣

Further Reading

The book [17] is an easy introduction to the main ideas of cryptography. Cryptography has a wonderfully long history that is worthy of study. One can learn much from the authoritative [11] and the delightful [21]. Cryptography is discussed in terms of complexity theory in Section 31.7 of [6], Section 10.6 of [22], Chapter 12 of [18], and Chapter 9 of [1].

Peter Hines has been doing some very intriguing work on the relationship between coherence theory and cryptography [9].

7.3 Kolmogorov Complexity Theory

Language can only deal meaningfully with a special, restricted segment of reality. The rest, and it is presumably the much larger part, is silence.
George Steiner
Page 21 of [24]

Kolmogorov complexity theory (sometimes also called **algorithmic information theory**) is about measuring the how much information a string has. We say the Kolmogorov complexity of string w is the size of the smallest Turing machine that can produce w. The idea is that if the string is a simple string then a small Turing machine can produce the string. In contrast, if the string is more complicated and has more information, then the Turing machine needs to be more complicated. What if the string is so complicated, that there are no small Turing machines that can produce it?

First some motivating examples. Consider the following three strings:

1. 000000000000000000000000000000000000000000000

2. 110111011111011111110111111111110111111111111110

3. 010100101101101010110111011110011000001111111010

All three are words in $\{0,1\}$, and are of length 45. It should be noted that if you flipped a coin 45 times the chances of observing any of these three sequences are equal. That is, the chances for each of the strings occurring is $\frac{1}{2^{45}}$. This demonstrates a fault of classical probability theory in measuring how much information a string has. Whereas you would not be shocked to see a sequence of coin flips produce a string like 3, the other two strings would be surprising.

Rather than looking at the probability of producing such a string, a better way of measuring the informational content is to look at the shortest programs that we can find to describe these strings:

1. `Print 45 0's.`

2. `Print the first 6 primes.`

3. `Print '010100101101101010110111011110011000001111111010'.`

The shorter the program, the less real information in the string. Such a string is "compressible" because rather then writing the whole string, you can

compress it and just write the program. In contrast, if only a long program can describe the string, then the string has more content.

What about the third string. There might be a shorter program that produces the third string, but I do not know of any. There could be some pattern in the string that I cannot see. We will see in Theorem 7.3.4, that there is no way to compute the size of the shortest program and so there is no way to tell if there is a shorter program.

If the only way to have a computer print a string is to literally have the string in the program then the string is "incompressible." An incompressible string is also called "random" because it has no patterns that we can use to print it out.

We note in passing that Kolmogorov complexity theory is not the only way to measure strings. There is complexity theory (how many steps does it take for the Turing machine to print the string), logical depth [2], sophistication [13], and others.

In order to formally describe Kolmogorov complexity we only need one part of one category from The Big Picture. We consider

$$\mathbb{T}\text{ot}\mathbb{T}\text{uring}(1,1) = Hom_{\mathbb{T}\text{ot}\mathbb{T}\text{uring}}(1,1). \tag{7.26}$$

This is the set of all Turing machines that accept one string and return one string. There is a size functor $Sz\colon \mathbb{T}\text{ot}\mathbb{T}\text{uring}(1,1) \longrightarrow \mathcal{N}$ where $\mathcal{N}$ is the total order category of natural numbers. This functor assigns to every total Turing machine the number of rules in the Turing machine.

Definition 7.3.1. Let x and y be strings. Then we define the **relative Kolmogorov complexity** to be the size of the smallest Turing machine that accepts input y, and outputs x. In symbols, $K\colon String \times String \longrightarrow Nat$ defined as

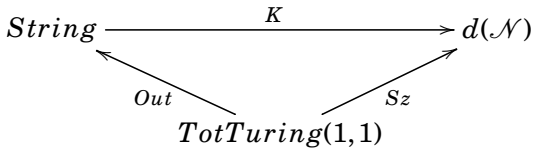
$$K(x|y) = \min_{\substack{T \in \mathbb{T}\text{ot}\mathbb{T}\text{uring}(1,1) \\ T(y)=x}} Sz(T) \tag{7.27}$$

If y is the empty string ϵ, then $K(x) = K(x|\epsilon)$ is the **Kolmogorov complexity** of x. This is the size of the smallest Turing machine that starts with an empty tape and outputs x. ◇

Technical Point 7.3.2. For those who know and love the language of Kan extension, the Kolmogorov complexity of a string can be given as a right Kan extension. Consider the functor

$$Out\colon \mathbb{T}\text{ot}\mathbb{T}\text{uring}(1,1) \longrightarrow String \tag{7.28}$$

that produces the output of a Turing machine when the empty string is given, i.e., $Out(T) = T(\epsilon)$. Then the right Kan extension



(7.29)

gives the Kolmogorov complexity functor. ♡

As we saw with the third string in our examples, we can always simply print out any string. In terms of Turing machines this means that there is a set of rules that can be combined with a string such that the Turing machine will start with an empty input tape and then print out the string. There will be one rule per character in the string. Let c be the size of the set of rules that work with any string. In terms of the third example above c is the size of `Print ' '`. We have just proven the following theorem.

Theorem 7.3.3. There exists a constant c such that for all strings x we have $K(x) \leq |x| + c$. ★

If $K(x) < |x|$ then x is **compressible**, otherwise x is **incompressible** and **random**. The use of the word random is counterintuitive. We usually thing of random as without any rules. Here we are saying that it needs so many rules that it can only be described by writing it out, i.e., random.

We end this short tour of Kolmogorov complexity theory with the main theorem about K. One might believe that K can be computed and we can find the exact amount of minimal structure each string contains. Wrong.

Theorem 7.3.4. $K : String \longrightarrow Nat$ is not in $\mathbb{T}\mathrm{ot}\mathbb{C}\mathrm{omp}\mathbb{F}\mathrm{unc}$, i.e., it is not a total computable function. ★

Proof. The proof is a proof by contradiction. Assume (wrongly) that K is a computable function. We will use this function to show a contradiction. If K is computable, then we can use K to compute the computable function $K' : Nat \longrightarrow String$ which is defined as follows:

1. Accept an integer n as input.
2. Go through every string $s \in \Sigma^*$ in lexicographical order
   a Calculate $K(s)$.
   b If $K(s) \leq n$, continue.
   c If $K(s) > n$, output s and stop.

For any n this program will output a string with a larger Kolmogorov complexity than n. This program has a size, say c. If we "hard-wire" a number n into the program, this would demand $\log n$ bits and the entire program will be of size $\log n + c$. Hard-wiring means making a computable function $K'' \colon * \longrightarrow Nat \longrightarrow String$ where $* \longrightarrow Nat$ picks out n. K'' will output a string that demands more complexity than n. Since we can find an n such that $n > \log n + c$, K'' will produce a string that has higher Kolmogorov complexity then the size of the Turing machine that produced it. This is a contradiction. Our assumption that K is computable is false. □

This deep theorem means we can never determine if there is more structure than what we see.

Advanced Topic 7.3.5. In 1975 Gregory J. Chaitin introduced [5] a number whose digits are not compressiable. The number which he called Ω is the probability that a random Turing machine will halt. If you look at descriptions of Turing machines and for every Turing machine T, you write $|T|$ as the size of the description, then Ω can be written as

$$\Omega = \sum_{T \ halts} 2^{-|T|} \tag{7.30}$$

In a sense, Ω has the unsolvablility of the Halting problem built into it. The number is uncomputable by any Turing machine and is transcendental (that means it is like π and e and cannot be described by algebraic means.) The Ω depends on how the Turing machines are described. Robert Solovay [23] formulated a way of describing Turing machines such that it is impossible to even determine one bit of Ω. See Section 7.1 of [4] and Section 3.6.2 of [14]. ○

Research Project 7.3.6. We used Kolmogorov complexity theory to define randomness. There are, however, other ways to describe randomness in terms of computability. Two of the other ways are Martin-Löf's randomness and randomness by constructive martingales. It is a fact that Kolmogorov randomness is equivalent to these other two methods. A nice research project would be to understand these ways of defining randomness. These other methods should also be described using the category of computable functions. It would be nice to then show that categorically these three methods define the same subset of the real numbers. There are still other, more powerful, ways to define randomness. Try to formulate them in terms of our categories of computable functions.

Another interesting project would be to describe other measures of string complexity like logical depth [2] and sophistication [13] . We described Kol-

mogorov complexity in a categorical way, it would be nice to see these other done in a categorical way. ♣

Further Reading

The main textbook in this field is [14]. Christian S. Calude's book [4] is wonderful. There is also a short, beautiful introduction to the whole field in Section 6.4 of [22]. One of the founders of this field is Gregory J. Chaitin. All of his books and papers are interesting and worth studying.

There is yet another connection between category theory and Kolmogorov complexity theory. In my [27] I extend the notion of Kolmogorv complexity from measuring strings to measuring categorical structures.

7.4 Algorithms

> *Of course if I am pinned down and asked to explain more precisely what I mean by these remarks, I am forced to admit that I don't know any way to define any particular algorithm except in a programming language.*
>
> Donald Knuth [12]

We close this text with a short discussion of the definition of "algorithm." Notice that this word has not been stressed so far. While we freely used the words "function," "program," "Turing machine," we avoided the word "algorithm." This is because the formal definition of the word is not simple to describe.

There are those who say that an algorithm is exactly the same thing as a program. In fact, on page 5 of the authoritative [6], an algorithm is informally defined as "any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output." We are left with asking what is a "procedure?" Furthermore, this informal definition seems like it is defining a program not an algorithm.

The problem with equating algorithms with programs is that the word "algorithm" is not used this way colloquially. If there are two programs that are very similar and only have minor differences, we usually do not consider them different algorithms. We say that the algorithm is the same but the programs are different. Here are some examples of two programs that are different but we would still consider to be the same algorithm:

- One program uses variable name x for a certain value, while the other program uses variable name y for the same value.
- One program performs a process n times in a loop, while another program performs the process $n-1$ times in a loop and then does the process one more time outside of the loop.
- One program performs two unrelated processes (they do not effect each other) in one order, while a second program performs the unrelated processes in the reverse order.
- One program performs two unrelated processes in one loop, while a second program performs each of the two unrelated processes in their own separate loop.

This list can easily be extended.

Let us make the case in another way. A teacher describes a certain algorithm to her computer class. She tells her thirty students to go home and implement the algorithm. Assuming that they are all bright and that there is no cheating, thirty *different* programs will be handed in the next class. Each program is an implementation of the algorithm. While there are differences among the programs, they are all "essentially the same." All the programs definitely implement the same function. This is the way that the word "algorithm" is used.

With this is mind, we make the following definition.

Definition 7.4.1. Take the set of all programs. We describe an equivalence relation on this set where two programs are equivalent if they are "essentially the same." An **algorithm** is an equivalence class of programs under this relation. Note that all the programs in the same equivalence class perform the same computable function. However there can be two different equivalence classes that also perform the same function. ◇

Figure 7.1 makes this all clear. There are three levels. The top level is the collection of all programs. The bottom level is the collection of all computable functions. And in-between them is the collection of algorithms. Consider programs $mergsort_a$ and $mergesort_b$. The first program is an implementation of mergesort programed by Alice, while the second program is written by Bob. They are both implementations of the algorithm $mergesort$ found in the middle level. There are also programs $quicksort_x$ and $quicksort_y$ that are different implementations of the algorithm $quicksort$. Both the algorithms $mergesort$ and $quicksort$ perform the same computable function $sort$. The big circle above the cone represented by $sort$ contains all the programs that implement the sort function. Above the computable function $find\ \ max$ there are all the programs that take a list and find the maximum element. Some of those programs are essentially the same and are implementations of the $binarysearch$ algorithm while others implement the $brute\ \ search$ algorithm.

Whenever we have equivalence classes, we have projection functors. (We can also discuss quotient categories). In terms of categories, this idea describes two full (symmetric monoidal) functors of (symmetric monoidal) categories.

$$\mathbb{P}\mathrm{rogram} \twoheadrightarrow \mathbb{A}\mathrm{lgorithm} \twoheadrightarrow \mathbb{C}\mathrm{ompFunc}$$

All the categories and bicategories have sequences of types as objects. $\mathbb{P}\mathrm{rogram}$ can be $\mathbb{T}\mathrm{uring}$, $\mathbb{R}\mathrm{egMach}$, and $\mathbb{C}\mathrm{irc}$ or any other syntactical way of describing computation. The left functor takes every program to the algorithm

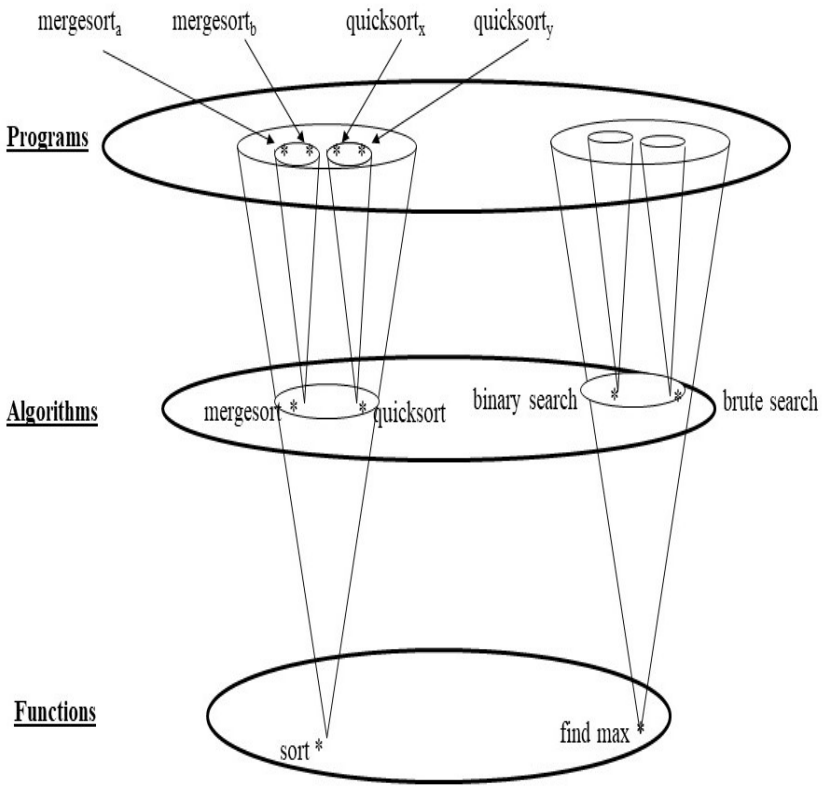


Figure 7.1: The Definition of an Algorithm

it implements. This functor is the identity on objects and full on morphisms. The right functor takes every algorithm to the computable function that it describes. It too is the identity on objects and full on morphisms.

The top level — programs — is the domain of programmers. The bottom level — computable functions — is really what mathematicians study. And the middle level is the core of computer science. The bicategories $\mathbb{P}\mathrm{rogram}$ (e.g., $\mathbb{T}\mathrm{uring}$, $\mathbb{R}\mathrm{egMach}$ and $\mathbb{C}\mathrm{irc}$, etc.) are syntactical in the sense that they are an exact description of a computational process. In contrast, $\mathbb{C}\mathrm{omp}\mathbb{F}\mathrm{unc}$ is semantic. The functions are the meaning of the programs. Algorithms are somewhat in-between syntax and semantics. They are "the ideas" of descriptions of functions.

Defining an object as an equivalence class of more concrete objects is not unusual. (i) Some philosophers follow Gottlob Frege in defining natu-

ral numbers as equivalence classes of finite sets that are bijective to each other. In detail, take the set of finite sets and put an equivalence relation: two sets are deemed equivalent if there exists a bijection between them. Every equivalence class corresponds to a natural number. (As category theorists, we say that the collection of natural numbers is the skeletal category of finite sets.) The number 3 is "implemented" by all the sets with three elements. (ii) Mathematicians describe a rational number as an equivalence class of pairs of integers. In detail, the pair (x, y) is equivalent to (x', y') if and only if $xy' = yx'$. The fraction $\frac{1}{3}$ is "implemented" by the pairs $(1,3)$, $(10,30)$, $(-30,-90)$, $(534,1602)$, etc. (iii) Physicists do not study physical phenomena. Rather, they study collections of phenomena. That is, they look at all phenomena and declare two phenomena to be equivalent if there is some type of symmetry between them. Two experiments that occur in different places, or are oriented differently, or occur at different times are considered the same if their outcome is the same. Laws of nature describe collections of physical phenomena, not individual phenomena. See [28] for more about this and the relationship between collections of phenomena and mathematics.

One can use these equivalence relations to discuss some very interesting category theory. There is something subjective about the question of when two programs are considered "essentially the same." Each possible answer will give us a different category of algorithms. (See [29] for more about this.) As we have said before, the set of programs does not form a category but a bicategory with extra structure. However, when we look at certain types of equivalence classes of programs we get certain categories with extra structure. The equivalence relations will determine which types of structure we will get. In a sense, the way we decide if two programs are "essentially the same" will correspond to coherence conditions. Going from programs to algorithms is a type of strictification. When we go further to genuine mathematical functions we are further strictifying and the coherence conditions are stronger. All this is done formally in [26, 29].

Research Project 7.4.2. It would be interesting if the functors from programs to algorithms and algorithms to functions had inverses or adjuncts. I seriously doubt that such things exist for all programs of the same power as Turing machines. (How would one even describe a function to make a functor from functions to algorithms? How would one even describe a an algorithm to make a functor from algorithms to programs. Also, if such functors existed there would be a program with universal properties for every function. This does not seem correct.)

However, one might lower their expectations and find some gems. For

the syntax level, rather then looking at full fledged Turing machines, look at finite automata. In that case, the semantic level will be regular languages. On the algorithms level, there will be regular expressions or Kleene algebras. In terms of funtors, there is a way of going from finite automaton to regular expression and vice versa. The Myhill–Nerode theorem can be used as a way of going from a regular language to a finite automaton. Advanced Topic 7.1.12 alludes to the universal properties of these functors. (See any formal language theory textbook for these topics.) ♣

Further Reading

The idea of defining an algorithm as an equivalence class of programs comes from my paper [26]. There is a followup to the paper which deals with many different equivalence classes [29]. The first paper was criticized by Andreas Blass, Nachum Dershowitz, and Yuri Gurevich in [3]. My definition of an algorithm is used in the second edition of Manin's logic book [16], and by several others since.

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: a modern approach*. Cambridge University Press, Cambridge, 2009.

[2] C. H. Bennett. Logical depth and physical complexity. In *A Half-century Survey on The Universal Turing Machine*, pages 227–257, New York, NY, 1988. Oxford University Press.

[3] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. When are two algorithms the same? *Bull. Symbolic Logic*, 15(2):145–168, 2009.

[4] Cristian S. Calude. *Information and randomness: an algorithmic perspective, Second edition, With forewords by Gregory J. Chaitin and Arto Salomaa*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, 2002.

[5] G. J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340, 1975.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms, Third edition*. MIT Press, Cambridge, MA, 2009.

[7] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, complexity, and languages*. Computer Science and Scientific Computing. Academic Press, Inc., Boston, MA, second edition, 1994.

[8] Ehrig H. et al. *Universal theory of automata. A categorical approach*. Teubner, 1974.

[9] Peter. Hines. Categorical coherence in cryptography, algebra, and number theory. *Submitted for publication*, 2018.

[10] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, Mass., 1979.

[11] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996.

[12] Donald E. Knuth. *Selected Papers on Computer Science*. Cambridge University Press, New York, NY, 1996.

[13] Moshe Koppel. Complexity, depth, and sophistication. *Complex Systems*, 1:1087–1091, 1987.

[14] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications, Third Edition*. Texts in Computer Science. Springer, New York, 2008.

[15] Saunders Mac Lane. *Categories for the working mathematician, 2nd edition*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1998.

[16] Yu. I. Manin. *A course in mathematical logic for mathematicians, Second Edition*, volume 53 of *Graduate Texts in Mathematics*. Springer, New York, 2010.

[17] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2009.

[18] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[19] Elaine Rich. *Automata, computability and complexity: theory and applications*. Pearson, New Jersey, 2008.

[20] Peter. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 124–134, Washington, DC, 1994. IEEE Computer Society.

[21] Simon Singh. *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography*. Doubleday, New York, NY, 1999.

[22] Michael Sipser. *Introduction to the Theory of Computation, Second edition*. Course Technology, 2006.

[23] Robert M. Solovay. A version of Ω for which ZFC can not predict a single bit. In C. S. Calude and G. Păun, editors, *Finite Versus Infinite. Contributions to an Eternal Dilemma*, pages 323–334. Springer, London, 2000.

[24] G. Steiner. *Language and Silence: Essays on Language, Literature, and the Inhuman*. Yale University Press, 1998.

[25] R. F. C. Walters. *Categories and computer science*, volume 28 of *Cambridge Computer Science Texts*. Cambridge University Press, Cambridge, 1991.

[26] Noson S. Yanofsky. Towards a definition of an algorithm. *J. Logic Comput.*, 21(2):253–286, 2011.

[27] Noson S. Yanofsky. Computability and complexity of categorical structures. *Available at http://www.sci.brooklyn.cuny.edu/noson/*, 2015.

[28] Noson S. Yanofsky. Why mathematics works so well. In *Trick or truth?*, Front. Coll., pages 145–156. Springer, Cham, 2016.

[29] Noson S. Yanofsky. Galois theory of algorithms. In *Rohit Parikh on logic, language and society*, volume 11 of *Outst. Contrib. Log.*, pages 323–347. Springer, Cham, 2017.

[30] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum computing for computer scientists*. Cambridge University Press, Cambridge, 2008.

Index