Title

A Thesis
Presented to
The Established Interdisciplinary Committee
for Mathematics and Computer Science
Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Riley Shahar

May 200x

Approved for the Committee
(Mathematics and Computer Science)

Angélica Osorno

Adam Groce

Table of Contents

Chapter 1

Cryptography

[TODO: An introduction to the chapter; cite [KL14; PS10; Ros21].]

1.1 Foundations

1.1.1 One-way functions

Many cryptographic protocols rely on *one-way functions*, which are informally functions that are easy to compute, but hard to invert. The former notion is easy to formalize in terms of time complexity, but the latter is more difficult. We typically ask that any "reasonably efficient" algorithm—called the *adversary*—attempting to invert the function has a negligible chance of success. (Recall that a function f is *negligible* if $f = o(n^{-k})$ for every k, in which case we write $f = \mathsf{negl}(n)$ or just $f = \mathsf{negl}$.)

Notation. We will use PPT as shorthand for probabilistic polynomial-time, and the term *adversary* for non-uniform PPT algorithms.

Definition 1.1 (one-way function)**.** A function f is *one-way* if:

- (easy to compute) f is PPT-computable;
- (hard to invert) for any adversary $\mathcal{A}$, natural number n, and uniform random choice of input x such that $|x| = n$,

$$\Pr[f(\mathcal{A}(1^n, f(x))) = f(x)] = \mathsf{negl}(n).$$

Note that $|x|$ here is *not* the absolute value, but is instead the length of x as a binary string: if x is a number, then by encoding in binary have that $|x| = \Theta(\log_2 x)$.

The idea is that, given $y = f(x)$, $\mathcal{A}$ attempts to find some x' such that $f(x') = y$. If some adversary can do this with non-negligible probability, then the function is not one-way. While the probability must be negligible in $|x|$, the adversary is given $f(x)$ and 1^n as an input, and hence must run polynomially only in $|f(x)| + n$. This is a common technique called *padding*, wherein algorithms are given an extra input of 1^n to ensure they have enough time to run.

We do not know that one-way functions exist. In fact, while the existence of one-way functions implies that P ≠ NP, the converse is not known[1]. However, as in the following examples, we have excellent candidates under fairly modest assumptions.

Example 1.2 (Factoring [PS10, subsection 2.3]). Suppose that for any adversary $\mathcal{A}$ and for uniform random choice of $x = pq$ for primes p and q,

$$\Pr[\mathcal{A}(x) = \{p, q\}] = \mathsf{negl}(\max\{|p|, |q|\}).$$

This is the *factoring hardness assumption*, for which there is substantial evidence. Then $(x, y) \mapsto xy$ is one-way.

Example 1.3 (Discrete Logarithm [KL14, subsection 8.3.2]). Let G be any fixed group. The *discrete logarithm hardness assumption* for G is that, for any adversary $\mathcal{A}$ and for uniform random choice of $g \in G$ and $h \in \langle g \rangle$ such that $h = g^k$,

$$\Pr[\mathcal{A}(g, h) = k] = \mathsf{negl}(|g|).$$

Under the discrete logarithm hardness assumption, $(g, k) \mapsto g^k$ is one-way.

The discrete logarithm hardness assumption is known to be false for certain groups, such as the additive groups $\mathbb{Z}_p$ for prime p, in which case $g^k = gk$ and the Euclidean algorithm solves the problem. However, it is believed to hold for groups such as $\mathbb{Z}_p^*$ for sufficiently big prime p. For a survey of various versions of this assumption, see [SS02].

1.1.2 Proofs by Reduction

Many cryptographic definitions, including Definition 1.1, take the form *for any adversary $\mathcal{A}$, natural number n, and uniform random choice of input x such that $|x| = n$, some predicate on the output of $\mathcal{A}$ has negligible probability.* The basic technique for proving results using these definitions is called *proof by reduction*. The idea is to reduce one problem into another by starting with an arbitrary adversary attacking the second and showing construct an adversary attacking the first, such that the probability of their successes is related. If we assume the first problem is hard, then by studying the structure of the reduction we can learn about the hardness of the second problem. As such, we often say that reductions prove *relative hardness results*, so that for instance Example 1.4 below proves the hardness of g relative to f.

More specifically, to prove hardness of a problem Π relative to Π', a proof by reduction generally goes as follows:

1. Fix an arbitrary adversary $\mathcal{A}$ attacking a problem Π.

2. Construct an adversary $\mathcal{A}'$ attacking a problem Π' which:

   (a) Receives an input x' to Π'.

   (b) Translates x' into an input x to Π.

[1] [Imp95] gives a classic discussion of the implications of various resolutions to P vs. NP on cryptography, including the case where P ≠ NP but one-way functions nevertheless do not exist.

(c) Simulates $\mathcal{A}(x)$, getting back an output y which solves $\Pi(x)$.

(d) Translates y into an output y' which solve $\Pi(x')$.

3. Analyze the structure of the translations to conclude that $\mathcal{A}'$ solves Π' with probability related to that with which $\mathcal{A}$ solves Π.

4. Given the hardness assumptions on Π', conclude relative hardness of Π.

The point is that $\mathcal{A}'$'s job is to "simulate" the problem Π to $\mathcal{A}$, using the data it gets from Π' to construct an input to Π. We illustrate this concept now.

Example 1.4 (a straightforward proof by reduction [PS10, subsection 2.4.1])**.** Let f be a one-way function. Then we claim $g : (x, y) \mapsto (f(x), f(y))$ is a one-way function. We can compute g in polynomial time by computing f twice, so it remains to show that g is hard to invert.

Let $\mathcal{A}$ be any adversary. We will construct an adversary $\mathcal{A}'$ such that, if $\mathcal{A}$ can non-negligibly invert g, then $\mathcal{A}'$ can non-negligibly invert f.

The adversary $\mathcal{A}'$ takes input 1^n and y. It then uniformly randomly chooses u of length n and computes $v = f(u)$, which is possible because f is easy to compute. Now $\mathcal{A}'$ computes $(u', x') := \mathcal{A}(1^{2n}, (v, y))$ and outputs x'.

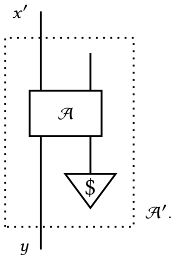
When $\mathcal{A}'$ simulates $\mathcal{A}$, it passes v, which is $f(u)$ for a uniform random u, and y, which is (on well-formed inputs) $f(x)$ for a uniform random x. Thus, this looks like exactly the input that $\mathcal{A}$ would "expect" to receive if it is attempting to break g. As such, whenever $\mathcal{A}$ successfully inverts g, $\mathcal{A}'$ successfully inverts f. Since everything is uniform we may pass to probabilities, and so:

$$\begin{aligned}
&\Pr[g(\mathcal{A}(1^{2n}, g(u, x))) = g(u, x)] \\
&\quad = \Pr[g(\mathcal{A}(1^{2n}, (f(u), f(x)))) = (f(u), f(x))] && \text{by definition of } g \\
&\quad \leq \Pr[f(\mathcal{A}'(1^n, f(x))) = f(x)] && \text{by the above argument} \\
&\quad = \mathsf{negl}(n) && \text{by the hardness assumption for } f.
\end{aligned}$$

Thus g is one-way.

Comparing this example to the above schema, we see that the problem Π' is to invert f, while the problem Π is to invert g. The input x' to Π' is y, while the computed input x to Π is (v, y). The output y of $\mathcal{A}$ is (x', u'), while the computed output y' is x'.

Diagramatically, we can represent the algorithm $\mathcal{A}'$ as follows:



While this is not standard notation in cryptography, it will be useful for our future purposes. We read these diagrams—called *circuit* or *string diagrams*—from bottom to top. This diagram says that $\mathcal{A}'$ is an algorithm which takes y, uniformly randomly generates another input (this is what the \$ means), calls $\mathcal{A}$, and returns its first output.

1.1.3 Computational Indistinguishability

Computational indistinguishability formalizes the notion of two probability distributions which "look the same" to adversarial processes. We begin with probability distributions, but because we want to do asymptotic analysis, we will eventually need to switch to working with sequences of probability distributions.

Definition 1.5 (computational advantage)**.** Let X and Y be probability distributions. The *computational advantage* of an adversary $\mathcal{D}$, called the *distinguisher*, over X and Y is

$$\mathrm{ca}_{\mathcal{D}}(X, Y) = \left| \Pr_{x \leftarrow X}[\mathcal{D}(x) = 1] - \Pr_{y \leftarrow Y}[\mathcal{D}(y) = 1] \right|.$$

The idea is that the distinguisher $\mathcal{D}$ is trying to guess whether its input was drawn from X or Y; the computational advantage is how often it can do so.

Proposition 1.6. *Let $\mathcal{D}$ be a fixed distinguisher. Then $\mathrm{ca}_{\mathcal{D}}$ is a pseudometric on the space of probability distributions over an underlying set A.*

Proof. Symmetry and non-negativity are immediate from the definition. To show the triangle inequality, let X, Y, and Z be probability distributions over A. Let

$$\hat{x} = \Pr_{x \leftarrow X}[\mathcal{D}(x) = 1],$$

and similarly for $\hat{y}$ and $\hat{z}$. Then,

$$\mathrm{ca}_{\mathcal{D}}(X, Z) = |\hat{x} - \hat{z}| \le |\hat{x} - \hat{y}| + |\hat{y} - \hat{z}| = \mathrm{ca}_{\mathcal{D}}(X, Y) + \mathrm{ca}_{\mathcal{D}}(Y, Z). \qquad \square$$

We now turn to the asymptotic case.

Definition 1.7 (probability ensemble)**.** A *probability ensemble* is a sequence $\{X_n\}$ of probability distributions.

We say that two ensembles are computationally indistinguishable if there is no efficient way to tell between them. Formally:

Definition 1.8 (computational indistinguishability)**.** Two probability ensembles $\{X_n\}$ and $\{Y_n\}$ are *computationally indistinguishable* if for any (non-uniform PPT) distinguisher $\mathcal{D}$ and any natural number n,

$$\mathrm{ca}_{\mathcal{D}}(X_n, Y_n) = \mathsf{negl}(n).$$

In this case, we write $\{X_n\} \overset{\mathrm{c}}{\equiv} \{Y_n\}$.

Remark 1.9. A natural thought is to define a metric on probability distributions by $\mathrm{ca}(X, Y) = \sup_{\mathcal{D}} \mathrm{ca}_{\mathcal{D}}(X, Y)$, and extend to ensembles by asking that $\mathrm{ca}(X_n, Y_n) = \mathsf{negl}(n)$. Unfortunately, this does not quite yield the correct notion, as there exist ensembles which are computationally indistinguishable, but have sequences of distinguishers whose advantages for any fixed n converge to 1.

Proposition 1.10. *Computational indistinguishability is an equivalence relation on the space of probability ensembles over a fixed set A.*

Proof. Reflexivity and symmetry follow from the case of distributions. To show transitivity, let $\{X_n\} \overset{\mathrm{c}}{\equiv} \{Y_n\}$ and $\{Y_n\} \overset{\mathrm{c}}{\equiv} \{Z_n\}$. Let $\mathcal{D}$ be any distinguisher. Then for any n,

$$\begin{aligned} \mathrm{ca}_{\mathcal{D}}(X_n, Z_n) &\leq \mathrm{ca}_{\mathcal{D}}(X_n, Y_n) + \mathrm{ca}_{\mathcal{D}}(Y_n, Z_n) && \text{by the triangle inequality} \\ &= \mathsf{negl}(n) + \mathsf{negl}(n) && \text{by assumption} \\ &= \mathsf{negl}(n). \end{aligned}$$

□

It is necessary to be precise about what is being claimed here. Transitivity states that for any *constant, finite sequence* of probability ensembles, if each is computationally indistinguishable from its neighbors, then the two ends of the sequence are computationally indistinguishable. In cryptography, we sometimes want to consider the more general case of a countable sequence of probability ensembles. We can do slightly better than the previous result:

Proposition 1.11. *Let $\{X^k\}$ be a sequence of probability ensembles, so that each $X^k = \{X^k_n\}$ is itself a sequence of probability distributions. Let $\{X^i\} \overset{c}{\equiv} \{X^{i+1}\}$ for each i. Let $\{Y_n = X_n^{K(n)}\}$ for some polynomial K. Then $\{X^1_n\} \overset{c}{\equiv} \{Y_n\}$.*

Proof. Let $\mathcal{D}$ be any distinguisher. Then for any n,

$$\begin{aligned} \mathrm{ca}_{\mathcal{D}}(X^1_n, Y_n) &= \mathrm{ca}_{\mathcal{D}}(X^1_n, X_n^{K(n)}) \\ &\leq \mathrm{ca}_{\mathcal{D}}(X^1_n, X^2_n) + \cdots + \mathrm{ca}_{\mathcal{D}}(X_n^{K(n)-1}, X_n^{K(n)}) \\ &= K(n)\mathsf{negl}(n) \\ &= \mathsf{negl}(n). \end{aligned}$$

In particular, the last equality follows because K is polynomial. □

On the other hand, the result does not hold for arbitrary K. As we will see, this is a fundamental limitation for cryptographic composition: we only expect composition to work up to polynomial bounds.

One more closure result is valuable:

Proposition 1.12. *Let $\{X_n\} \overset{c}{\equiv} \{Y_n\}$, and let $\mathcal{M}$ be a non-uniform PPT algorithm. Then $\{\mathcal{M}(X_n)\} \overset{c}{\equiv} \{\mathcal{M}(Y_n)\}$.*

Proof. The proof is by reduction. Let $\mathcal{D}$ be a distinguisher. Then construct $\mathcal{D}'$ which, on input x, simulates $\mathcal{D}(\mathcal{M}(x))$. Then $\mathcal{D}'$ outputs 1 on x if and only if $\mathcal{D}$ outputs 1 on $\mathcal{M}(x)$, so

$$\mathrm{ca}_{\mathcal{D}}(\mathcal{M}(X_n), \mathcal{M}(Y_n)) = \mathrm{ca}_{\mathcal{D}'}(X_n, Y_n) = \mathsf{negl}(n)$$

by the computational indistinguishability assumption. □

1.2 Composition

Chapter 2

Category Theory

The notion of a *category*, originally developed as an abstraction for certain ideas in pure mathematics, turns out to be the natural algebraic axiomatization of a collection of strongly typed, composable processes, such as functions in a strongly typed programming language. More philosophically, we can think of a category as an *algebra of composition*, and category theory as the mathematical study of composition. In this chapter, we will develop the basic theory of categories, prioritizing examples from computer science where possible.

Basic texts on category theory include [Mac71] and [Rie17], while the connection to computer science is explored in [Pie91] and [BW90]. A more advanced treatment of the connection, especially applications to programming language theory, is [Jac99].

2.1 Basic Notions

2.1.1 Categories

Definition 2.1 (category)**.** A *category* C consists of the following data:

- a collection[1] of objects, overloadingly also called C;
- for each pair of objects $x, y \in C$, a collection of *morphisms* $C(x, y)$;
- for each object $x \in C$, a designated *identity morphism* $x \xrightarrow{1_x} x$;
- for each pair of morphisms $x \xrightarrow{f} y \xrightarrow{g} z$, a designated *composite morphism* $x \xrightarrow{gf} z$.

This data must satisfy the following axioms:

- *unitality*: for any $x \xrightarrow{f} y$, $1_y f = f = f 1_x$;
- *associativity*: for any $x \xrightarrow{f} y \xrightarrow{g} z \xrightarrow{h} w$, $(hg)f = h(gf)$.

Notation. In addition to those used above, many syntaxes are common for basic categorical notions.

[1] We use the word *collection* for foundational reasons: in many important examples, the objects and morphisms do not form sets. We ignore such foundational issues here; they are discussed in [Mac71, subsection 1.6].

- A morphism $f \in C(x, y)$ is often written $f\colon x \longrightarrow y$ or $x \xrightarrow{f} y$; x is called its *domain* or *source* and y is called it *codomain* or *target*.
- Morphisms may be called maps, arrows, or homomorphisms; the class of morphisms $C(x, y)$ may also be written $\mathrm{Hom}_C(x, y)$ or just $\mathrm{Hom}(x, y)$, and is often called a *hom-set*.
- Composition is written gf or $g \circ f$, or sometimes in the left-to-right order fg.
- Identities are written 1_x, id_x, or just x where the context is clear[2].

Example 2.2 (functional programming languages)**.** Consider some strongly-typed functional programming language L, whose functions are never side-effecting. Then under very modest assumptions about L, we can make a category $\mathcal{L}$, as follows:

- the objects of $\mathcal{L}$ are the types of L;
- the morphisms $\mathcal{L}(A, B)$ are the functions of type $A \longrightarrow B$;
- the identities 1_A are the identity functions $A \longrightarrow A$;
- composition of morphisms are the usual function composition.

If L is truly non-side-effecting, then it's straightforward to check that this construction does indeed satisfy the axioms of a category; see for instance [BW90, subsection 2.2] to see the necessary assumptions spelled out rigorously.

Categories are also widespread in mathematics, as the following examples show.

Example 2.3 (concrete categories)**.** The following are all categories:

- Set is the category of sets and functions.
- Grp is the category of groups and group homomorphisms.
- Ring is the category of rings and ring homomorphisms.
- Top is the category of topological spaces and homeomorphisms.
- For any field $\Bbbk$, $\mathrm{Vect}_{\Bbbk}$ is the category of vector spaces over $\Bbbk$ and linear transformations.

We call such categories, whose objects are structured sets and whose morphisms are structure-preserving set-functions, *concrete*. On the other hand, many categories look quite different.

Example 2.4. The following are also categories:

- The *empty category* has no objects and no morphisms.
- The *trivial category* has a single object and its identity morphism.
- Any group (or, more generally, monoid) can be thought of as a category with a single object, a morphism for every element, and composition given by the monoid multiplication.
- Any poset (or, more generally, preorder) $(P, \leq)$ can be thought of as a category whose objects are the elements of P, with a unique morphism $x \longrightarrow y$ if and only if $x \leq y$. In this sense, composition is a "higher-dimensional" transitivity, and identities are higher-dimensional reflexivity.

[2] I agree with Harold Simmons, who says that this last is "a notation so ridiculous it should be laughed at in the street" [Sim11, p. 5].

- Associated to any directed graph is the *free category* on the graph, whose objects are nodes and whose morphisms are paths. In particular, the identities are just the empty paths, while composition concatenates two paths.
- Let $M = (Q, \delta)$ be an automaton over an alphabet Σ, so that $\delta : Q \times \Sigma \to Q$ is a transition function (one may replace Q with $\mathcal{P}(Q)$ in the codomain to represent a nondeterministic automaton). There is an associated category $\mathcal{M}$ whose objects are exactly the states and whose morphisms $\mathcal{M}(q_1, q_2)$ are the words $w \in \Sigma^*$ such that, if M is in the state q_1 and receives w as input, it ends in the state q_2. The identity morphism 1_q is the empty word, and composition is concatenation of words[3].
- There is a category whose objects are (roughly) multisets of molecules and whose morphisms are chemical reactions. See [BP17] for a formalization of this notion.

One more example will be critical for our purposes.

Example 2.5 (Categories of $\mathbb{A}$-computable maps)**.** Let $\mathbb{A}$ be a class of algorithms which is closed under composition. Then we can define a category whose objects are finite sets and whose morphisms are the $\mathbb{A}$-computable maps between them. The most important examples for our purposes are:

- the category Poly of finite sets and polynomial-time computable maps;
- the category PPT of finite sets and probabalistic polynomial-time computable stochastic maps;
- the category NUPPT of finite sets and non-uniform probabalistic polynomial-time computable stochastic maps.

When working with categories, we often want to show that two complex composites equate. In this case, we prefer graphical notation to the more traditional symbolic equalities of Definition 2.1. A diagram in a category C looks something like so[4]:

$$\begin{tikzcd} w \arrow[r, "f"] \arrow[d, "h"'] & x \arrow[d, "g"] \\ y \arrow[r, "k"'] & z. \end{tikzcd}$$

This diagram identifies four objects $w, x, y, z \in C$, and four morphisms $f \in C(w, x)$, $g \in C(x, z)$, $h \in C(w, y)$, and $k \in C(y, z)$.

We say that a diagram *commutes* if, for any pair of paths through the diagram with the same start and end, the composite morphisms are equal. In this language, the previous diagram commutes if and only if $gf = kh$.

Example 2.6. The axioms of Definition 2.1 are expressed by commutativity of the fol-

[3] I believe this example is due to [Gog+73, Example 2.2].

[4] The notion of a diagram can be made precise fairly easily; see [Rie17, subsection 1.6].

lowing three diagrams:

$$\begin{tikzcd} x \arrow[r, "f"] \arrow[rr, bend left, "gf"] & y \arrow[r, "g"] \arrow[rr, bend right, "hg"'] & z \arrow[r, "h"] & w \end{tikzcd} \qquad \begin{tikzcd} x \arrow[r, "1_x"] \arrow[dr, "f"'] & x \arrow[d, "f"] \\ & y \end{tikzcd} \qquad \begin{tikzcd} x \arrow[r, "f"] \arrow[dr, "f"'] & y \arrow[d, "1_y"] \\ & y. \end{tikzcd}$$

The key idea is that commutative diagrams can be "pasted", allowing us to build up complex equalities from simpler ones. For instance, if

$$\begin{tikzcd} w \arrow[r, "f"] \arrow[d, "h"'] & x \arrow[d, "g"] \\ y \arrow[r, "k"'] & z \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} x \arrow[r, "l"] \arrow[dr, "g"'] & v \arrow[d, "m"] \\ & z \end{tikzcd}$$

both commute, then by pasting along the shared morphism g, so does

$$\begin{tikzcd} w \arrow[r, "f"] \arrow[d, "h"'] & x \arrow[r, "l"] & v \arrow[d, "m"] \\ y \arrow[rr, "k"'] & & z. \end{tikzcd}$$

This pasting property is essentially just a re-expression of the transitivity and substitution properties of equality, but gives an extraordinarily useful geometric intuition to categorical arguments.

2.1.2 Functors

The philosophy of category theory is that

to study an object, one should study its morphisms.

Since we now want to study categories, we ask the natural question: what is the right notion of morphism between categories? The answer is a *functor*, which is just a structure-preserving map between categories.

Definition 2.7 (functor). A *functor* $F : \mathcal{C} \to \mathcal{D}$ consists of the following data:

- for each object $x \in \mathcal{C}$, an object $Fx \in \mathcal{D}$;
- for each morphism $f \in \mathcal{C}(x, y)$, a morphism $Ff \in \mathcal{D}(Fx, Fy)$.

This data must preserve the structure of the category, namely identities and composites, meaning:

- for each object $x \in \mathcal{C}$, $F1_x = 1_{Fx}$;
- for each pair of morphisms $x \xrightarrow{f} y \xrightarrow{g} z$ in $\mathcal{C}$, $F(gf) = (Fg)(Ff)$.

Example 2.8. In mathematics, functors are ubiquitous as representations of procedures for producing structures of one sort from structures of another. For instance, the following are all functors:

- On any category C, there is an *identity functor* $1_C : C \to C$ which takes each object and morphism to itself.
- There is a functor $\mathcal{P}_\exists : \textsc{Set} \to \textsc{Set}$ which takes a set X to its powerset, and a set-function $f : X \to Y$ to the direct image map given by

  $$f_\exists(A) = \{y \in Y : \exists a \in A \text{ such that } y = f(a)\}.$$

- There is a distinct functor $\mathcal{P}_\forall : \textsc{Set} \to \textsc{Set}$ which takes a set X to its powerset, and a set-function $f : X \to Y$ to the map given by

  $$f_\forall(A) = \{y \in Y : \forall x \in X, f(x) = Y \text{ implies } x \in A\}.$$

  As these examples show, the action of a functor on morphisms is not determined by its action on objects. (In fact, as usual in category theory, it is the action on morphisms—in particular, on the identities—which determines the action on objects.)
- There is a functor `List` : $\textsc{Set} \to \textsc{Set}$ which takes a set X to the set of all finite lists of elements in X, and a set-function f to its mapping over lists, i.e.

  $$(\texttt{List}f)([x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)].$$

  In other contexts, this functor is also called the *free monoid* or the *Kleene star.*
- For any field $\Bbbk$, there is a functor $\textsc{Set} \to \textsc{Vect}_\Bbbk$ which takes a set X to the $\Bbbk$-span of X, and a set-function f to its linear extension. This is also called the *free vector space.* More generally, any free construction—such as the free group, free ring, etc.—forms a functor.
- Let C be a concrete category, such as those of Example 2.3. Then the *forgetful functor* $U : C \to \textsc{Set}$ takes each object to its underlying set, and each morphism to its underlying set-function, "forgetting" the additional structure.
- There is also a forgetful functor $\textsc{Ring} \to \textsc{Grp}$ which takes each ring to its underlying additive group, and each ring homomorphism to its underlying group homomorphism.

Example 2.9. As the following examples show, whenever we can think of each instance of a certain mathematical structure as a category, functors reproduce the right notion of structure-preserving transformation between those structures.

- Let P and Q be posets with associated categories $\mathcal{P}$ and $\mathcal{Q}$. Let $p_1 \leq_P p_2$, so that there is a unique morphism $p_1 \to p_2$ in $\mathcal{P}$. Since F must take this morphism to a morphism $Fp_1 \to Fp_2$, it must hold that $Fp_1 \leq_Q Fp_2$. Furthermore, this is the only requirement on functors, as the statements about identities and composites assert equalities between morphisms, but any two morphisms with the same domain and codomain are equal in a poset. As such, functors between posets are exactly monotone maps.

- Let G and H be groups with associated categories $\mathcal{G}$ and $\mathcal{H}$. A functor $F : \mathcal{G} \to \mathcal{H}$ assigns the single object of $\mathcal{G}$ to the single object of $\mathcal{H}$, and each morphism in $\mathcal{G}$, which is an element $g \in G$, to a morphism (element) $Fg \in H$. That this preserves composites tells us that it preserves group multiplication, and hence it is a homomorphism. The fact that F preserves identities is extraneous, since every group homomorphism preserves identities. As such, functors between groups are exactly group homomorphisms.
- Let L_1 and L_2 be functional programming languages with associated categories $\mathcal{L}_1$ and $\mathcal{L}_2$. We think of a functor $F : \mathcal{L}_1 \to \mathcal{L}_2$ as an embedding—or, more technically, a *model*—of $\mathcal{L}_1$ in $\mathcal{L}_2$. Specifically, for any function in $\mathcal{L}_1$, F identifies a corresponding function in $\mathcal{L}_2$, and so F allows us to think of computations in L_2 as "simulating" computations in L_1.

If functors are morphisms between categories, then we should expect that there is a category of categories. This is indeed the case, but we first need to show that functors can be composed.

Proposition 2.10. *Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ be functors. Then there is a* composite functor *$GF : \mathcal{C} \to \mathcal{E}$, defined by $(GF)x = G(Fx)$ and $(GF)f = G(Ff)$.*

Example 2.11. The composite of the forgetful functors Ring → Grp and Grp → Set is exactly the forgetful functor Ring → Set.

Definition 2.12. The *category of categories* Cat has categories as objects and functors as morphisms.

The foundationally-inclined reader will correctly object to this definition, which implies that Cat should be an object of itself, leading to issues involving Russell's paradox. There are several resolutions to this—for instance, letting Cat be the category of so-called *locally small* categories, whose hom-sets $\mathcal{C}(x, y)$ each form sets. We ignore these issues here.

2.1.3 Natural Transformations

2.2 Monoidal Categories

In ordinary categories, composition is sequential: if morphisms are interpreted as computational processes, the composite gf means roughly "first do f, then do g." In many settings, we want to consider both sequential and parallel composition. The categorical axiomatization of this idea is *monoidal categories.*

2.2.1 The Definition

To model parallel composition, we want an binary operation $\otimes$ which assigns, to each pair of processes (morphisms) $f : x \to y$ and $g : w \to z$, their parallel composite $f \otimes g$. If we think of objects as types, this parallel composite can only run given inputs of both types x and w, to feed to f and g respectively, and should produce two outputs of types y and z. To represent this notion, we also need a way to pair types (objects), which means a binary operation also called $\otimes$ on morphisms. This dual assignment on both objects and morphisms suggests functoriality: we will ask that $\otimes$ is a functor $C \times C \to C$.

What axioms should this data satisfy? As in most well-behaved algebraic structures, there should be an identity for $\otimes$ on objects, which we will write I. Computationally, we may think of I as a "trivial resource," which may freely be created and has no uses. This I induces an identity, the morphism 1_I, for $\otimes$ on morphisms, so we do not need to add an identity on morphisms as an extra axiom. We would also parallel composition to associate, so that we can sensibly talk about performing n processes in parallel. It is therefore tempting to list the following axioms:

$$I \otimes x = x = x \otimes I; \qquad (x \otimes y) \otimes z = x \otimes (y \otimes z).$$

While this notion, called a *strict monoidal category*, is useful, it is not the most natural axiomatization. For instance, even the category Set, with the ordinary Cartesian product, is not strictly monoidal: the identity is $\{*\}$, but $\{*\} \times X$ is not equal to X, instead merely isomorphic. The point is that there is interesting structure in the way that even isomorphic objects relate to each other; we do not want to lose it by forcing strict equality.

However, we do not want to allow the strcture of these natural isomorphisms to be too strange. For instance, one can imagine two ways to convert from $I \otimes (x \otimes y)$ to $x \otimes y$:

$$I \otimes (x \otimes y) \cong x \otimes y \quad \text{and} \quad I \otimes (x \otimes y) \cong (I \otimes x) \otimes y \cong x \otimes y.$$

The first directly uses unitality, while the second associates and then uses unitality. A *coherence axiom* asserts that choices like this do not matter: every pair of comopsites of our canonical isomorphisms with the same domain and codomain should commute.

We are not quite ready; there is one remaining technical issue, though this paragraph may be safely skipped. It may happen that two domains equate "accidentally", so that, for instance,

$$((x \otimes y) \otimes z) \otimes w = x \otimes (y \otimes (z \otimes w)). \tag{2.1}$$

In this case, the version of the coherence axiom stated above implies that the isomorphisms

$$((x \otimes y) \otimes z) \otimes w \cong (x \otimes y) \otimes (z \otimes w) \quad \text{and} \quad x \otimes (y \otimes (z \otimes w)) \cong (x \otimes y) \otimes (z \otimes w)$$

should commute; they do, after all, have the same domain and codomain. But the first re-associates from the left to the right, and the second re-associates from the right to the left: these are structurally different actions, which only "look the same" because of the accident of Equation 2.1, so our theory should not require them to commute. There is a way to formalize a correct abstract notion of coherence—see for instance [Mac71, subsection VII.2]—but fortunately, Mac Lane's *coherence theorem* enables an easier axiomatization.

We are finally now ready to state the definition of a monoidal category.

Definition 2.13 (monoidal category)**.** A *monoidal category* C consists of the following data:

- an underlying category C;
- a functor $\otimes : C \times C \to C$, called the *tensor product*;
- an object $I \in C$, called the *tensor unit*;
- a natural isomorphism $\alpha_{x,y,z} : (x \otimes y) \otimes z \to x \otimes (y \otimes z)$, called the *associator*;
- a natural isomorphism $\lambda_x : I \otimes x \to x$, called the *left unitor*[5];
- a natural isomorphism $\rho_x : x \otimes I \to x$, called the *right unitor*.

This data must make the following diagrams, called the *triangle* and *pentagon* identities, commute:

$$\begin{tikzcd}
(x \otimes I) \otimes y \arrow[rr, "\alpha_{x,1_\otimes,y}"] \arrow[dr, "\rho_x"'] & & x \otimes (I \otimes y) \arrow[dl, "\lambda_x"] \\
& x \otimes y &
\end{tikzcd}$$

$$\begin{tikzcd}
& (x \otimes y) \otimes (z \otimes w) \arrow[dr, "\alpha_{x,y,z \otimes w}"] & \\
((x \otimes y) \otimes z) \otimes w \arrow[ur, "\alpha_{x \otimes y,z,w}"] \arrow[d, "\alpha_{x,y,z} \otimes 1_w"'] & & x \otimes (y \otimes (z \otimes w)) \\
(x \otimes (y \otimes z)) \otimes w \arrow[rr, "\alpha_{x,y \otimes z,w}"'] & & x \otimes ((y \otimes z) \otimes w). \arrow[u, "1_x \otimes \alpha_{y,z,w}"']
\end{tikzcd}$$

The above diagrams look arbitrary, but as mentioned, they are exactly what is required for the correct notion of coherence. On first exposure to these ideas, it is safe to ignore the exact statement of the identities and work with the intuition that any two ways of associating or unitalizing should be the same.

In the above definition, the natural isomorphisms α, λ, and ρ feel in some sense more like axioms than data. This is another key component of the category-theoretic philosophy, one which should feel comfortable to computer scientists, who often assume the existence of concrete objects which structure our models:

structure is a kind of data.

If we think of categories as algebras of structure, it is natural that we should think of axiomatic structure as an algebraic object which may be manipulated[6].

[5] The letters λ and ρ are chosen for their association with L and R, respectively.

[6] Of course, Definition 2.13 still carries a traditional-looking equational theory in the form of the triangle and pentagon identities. The key difference is that this theory is an assumption about the "two-dimensional" structure of the natural transformations, whereas associativity and unitality are assumptions about the "one-dimensional" structure of the functor $\otimes$. We could continue to generalize, instead asking that these diagrams are themselves witnessed by "three-dimensional" isomorphisms between the natural isomorphisms α, λ, and ρ. Repeating this process *ad infinitum*, the natural endpoint of the structure-as-data philosophy is so-called ∞-*category theory*.

2.2.2 Examples

The notion of a monoidal category is quite general; we survey some important examples here.

Example 2.14. Let us very explicitly construct the required data to show that Set is a monoidal category under the Cartesian product. The tensor unit is the singleton $\{*\}$. The associator is the natural isomorphism with components

$$\begin{aligned}\alpha_{X,Y,Z}\colon (X\times Y)\times Z &\to X\times(Y\times Z)\\ ((x,y),z) &\mapsto (x,(y,z)).\end{aligned}$$

The left and right unitors are the natural isomorphism with components

$$\begin{aligned}\lambda_X\colon \{*\}\times X &\to X\\ (*,x) &\mapsto x,\end{aligned}\qquad\qquad \begin{aligned}\rho_X\colon X\times\{*\} &\to X\\ (x,*) &\mapsto x.\end{aligned}$$

A common complaint about category theory is at play here: we now have a large number of relationships to demonstrate, including functorality of ×, naturality of α, λ, and ρ, and the pentagon and triangle identities. The author's opinion is that this work will ultimately save effort, by allowing us to use a powerful abstract theory across any structure we have shown to be monoidal, but if the reader is not convinced, one solution is to work even more generally. For instance, by showing that the Cartesian product satisfies a simple property called the *universal property of the product*, we could automatically conclude on the grounds of a general theorem that it is monoidal. Abstraction of this sort ultimately saves effort, but it is not always comfortable at first. Regardless, in order to exemplify the definition in all its detail, we continue with the explicit demonstration.

To show functorality of ×, we need to determine its action on morphisms. Letting $f : X \to Y$ and $g : W \to Z$, we define

$$\begin{aligned}f\times g\colon X\times W &\to Y\times Z\\ (x,y) &\mapsto (f(x),g(y)).\end{aligned}$$

This is functorial: it takes an identity $1_{(X,W)} = (1_X, 1_W)$ to $1_{X\times W}$, and the composite of two pairs of morphisms to composite of their action on pairs.

To show naturality of α, let $(f,g,h) : (X,Y,Z) \to (X',Y',Z')$ be a morphism in Set^3. We need to show that the following diagram commutes:

$$\begin{tikzcd}
(X\times Y)\times Z \arrow[r, "\alpha_{X,Y,Z}"] \arrow[d, "(f\times g)\times h"'] & X\times(Y\times Z) \arrow[d, "f\times(g\times h)"] \\
(X'\times Y')\times Z' \arrow[r, "\alpha_{X',Y',Z'}"'] & X'\times(Y'\times Z').
\end{tikzcd}$$

Tracking the action of a triple $((x, y), z)$ through both paths, we see the needed equality:

$$\begin{tikzcd}
((x,y),z) \arrow[r, mapsto, "\alpha_{X,Y,Z}"] \arrow[d, mapsto, "(f\times g)\times h"'] & (x,(y,z)) \arrow[d, mapsto, "f\times(g\times h)"] \\
((f(x),g(y)),h(z)) \arrow[r, mapsto, "\alpha_{X',Y',Z'}"'] & (f(x),(g(y),h(z))).
\end{tikzcd}$$

To show naturality of λ, let $f : X \to Y$. Since the only morphism $\{*\} \to \{*\}$ is $1_{\{*\}}$, naturality is entailed by commutativity of the following diagram:

$$\begin{tikzcd}
\{*\}\times X \arrow[r, "\lambda_X"] \arrow[d, "1_{\{*\}}\times f"'] & X \arrow[d, "f"] & & (*,x) \arrow[r, mapsto, "\lambda_X"] \arrow[d, mapsto, "1_{\{*\}}\times f"'] & x \arrow[d, mapsto, "f"] \\
\{*\}\times Y \arrow[r, "\lambda_Y"'] & Y, & \text{i.e.} & (*,f(x)) \arrow[r, mapsto, "\lambda_Y"'] & f(x).
\end{tikzcd}$$

Naturality of ρ is similar. We show the pentagon identity by its action on $(((x, y), z), w)$:

$$\begin{tikzcd}
 & ((x,y),(z,w)) \arrow[dr, mapsto, "\alpha_{X,Y,Z\times W}"] & \\
(((x,y),z),w) \arrow[ur, mapsto, "\alpha_{X\times Y,Z,W}"] \arrow[d, mapsto, "\alpha_{X,Y,Z}\times 1_W"'] & & (x,(y,(z,w))) \\
((x,(y,z)),w) \arrow[rr, mapsto, "\alpha_{X,Y\times Z,W}"'] & & (x,((y,z),w)). \arrow[u, mapsto, "1_X\times\alpha_{Y,Z,W}"']
\end{tikzcd}$$

The triangle identity is similar.

While we will never again be so explicit, we hope the previous example makes the axioms of a monoidal category more concrete.

Example 2.15. There are many more examples of monoidal categories throughout mathematics.

- Vect$_{\mathbb{k}}$ is monoidal with the tensor product of vector spaces.
- Cat is monoidal with the product category.
- The categories of Example 2.5 are monoidal with the Cartesian product of sets. In particular, the cartesian product of two $\mathbb{A}$-computable functions is computable by an algorithm which simply first computes the first function, and then computes the second function.
- Let L be a strongly-typed functional programming language with *product types*, which means that for any types A and B, there is a type $A \times B$ whose elements are pairs (a, b) of elements $a \in A$ and $b \in B$. Then the category $\mathcal{L}$ is monoidal in the same way as Set.

Example 2.16 (concurrent programming [MM90])**.** Returning to our motivation of parallelism, here is a very different example. Let L be a strongly-typed functional *concurrent* programming language, by which we mean that it can run computations concurrently on different machine threads. Then again under reasonable assumptions, $\mathcal{L}$ is monoidal, with concurrent branching as the tensor product and the do-nothing program as the tensor unit.

2.2.3 String Diagrams

In monoidal categories, there are two "formal mechanisms" for building morphisms: sequential composition $\circ$ and parallel composition $\otimes$. String diagrams are a graphical calculus for morphisms using these mechanisms. String diagrams and related calculi are explored in great detail by [Sel11]; we give a basic outline here.

Consider a monoidal category C with three morphisms $f : x \to y$, $g : w \to z$, and $h : y \otimes z \to u$. We can can form a new morphism $h \circ (f \otimes g) : x \otimes w \to u$. We encode this fact in the following *string diagram*:

2.2.4 Commutativity: Braiding and Symmetry

While monoidal categories are necessarily associative, nothing in the definition guarantees that the monoidal product is commutative. To enforce symmetry, we add a natural isomorphism $\gamma_{x,y} : x \otimes y \to y \otimes x$, called the *braiding*, to the data. There are, however, two different possible versions of the coherence identities.

2.2.5 Monoidal Functors

2.2.6 Multicategories

2.3 Resource Theories

Appendix A

Computer Scientific Foundations

In the main body, we have assumed standard material from a course in computability and complexity, including function asymptotics, the notion of an algorithm, and the complexity class P. We briefly overview these ideas here; a standard text is [Sip13].

A.1 Asymptotics

Function asymptotics formalize the notion of a function approximating another function. In particular, for a pair of functions $f, g : \mathbb{N} \to \mathbb{R}$, we often want to compare f and g on large inputs and only up to a constant factor. This is most common in runtime analysis, the idea being that the running time of algorithms on small inputs is less important to their overall performance than their running time on large inputs. We formalize this notion as follows:

Definition A.1 (Function Asymptotics)**.** Let $f, g : \mathbb{N} \to \mathbb{R}$ be a pair of functions which are both non-negative for sufficiently large inputs. We say that f is *big-Oh of g*, written $f = O(g)$, if there exists a constant $c > 0$ such that for all n sufficiently large,

$$cf(n) \geq g(n).$$

In this case, we also say that g is *big-Omega of f*, written $g = \Omega(f)$.

If $f = O(g)$ and $g = O(f)$, we say that f is *big-Theta of g*, written $f = \Theta(g)$. Explicitly, this means that there exist constants $c_1, c_2 > 0$ such that for all n sufficiently large,

$$c_1 f(n) \leq g(n) \leq c_2 f(n).$$

If $f = O(g)$ but $f \neq \Theta(g)$, we say that f is *little-oh of g*, written $f = o(g)$, and g is *little-omega of f*, written $g = \omega(f)$. Explicitly, this means that for all constants $\epsilon > 0$ and all n sufficiently large,

$$f(n) \leq \epsilon g(n).$$

Notation. By abuse of notation, we often write $f(n) = O(g(n))$ to mean that f is $O(n)$; for example, the statement that n^2 is $O(n^3)$ means that the function $f(n) = n^2$ is $O(g)$, where g is the function $n \mapsto n^3$.

Example A.2. We have that:

- $17n^2$ is $o(n^3)$, $\omega(n)$, and $\Theta(n^2)$;
- $\log n$ is $o(n)$, $\omega(1)$, and $\Theta(\ln n)$;
- e^n is $\omega(n^k)$ for any exponent k;
- e^{-n} is $o(n^{-k})$ for any exponent k.

These last two examples are especially important. No matter how big the power, an exponential will always dominate a polynomial for sufficiently big n. Because of the importance of polynomials in theoretical computer science, we say a function f is *negligible* if $f = o(n^k)$ for all k. In this case, we write $f = \mathsf{negl}(n)$ or just $f = \mathsf{negl}$.

Proposition A.3. *Big-Oh is a preorder on the set of functions $\mathbb{N} \longrightarrow \mathbb{N}$. The induced equivalence relation is exactly big-Theta.*

In the partial order of equivalence classes under Θ, O behaves like $\leq$, o like $<$, and Θ like $=$. As suggested by the notation $f = O(g)$, it is common in some contexts to treat functions as identical with their asymptotic equivalence class.

Proposition A.4. *Let $f_1 = O(g_1)$ and $f_2 = O(g_2)$. Let c be any nonzero constant. Then,*

$$f_1 + f_2 = O(\max\{g_1, g_2\}), \quad cf_1 = O(g_1), \quad \text{and} \quad f_1 f_2 = O(g_1 g_2).$$

In other words,

$$O(g_1) + O(g_2) = O(\max\{g_1, g_2\}), \quad cO(g) = O(g), \quad \text{and} \quad O(g_1)O(g_2) = O(g_1 g_2).$$

Identical results hold for o and Θ.

Proposition A.4 justifies the universal practice of dropping constants and small additive terms from asymptotics, so that for instance $n^2 + n + \ln n = \Theta(n^2)$.

A.2 Algorithms and Determinism

Our basic notion is of an *algorithm* over a finite alphabet Σ, usually $\{0, 1\}$. An algorithm $\mathcal{A}$ is intuitively some set of steps which take an input word x over Σ, perform some transformations, and output another word $\mathcal{A}(x)$ over Σ. An algorithm may have certain *side effects*, such as sending a message or logging a string, and its behavior may not be deterministic. There are several ways to formalize the notion of algorithm—most common in cryptography are Turing machines—but we will not need to be so precise here.

Algorithms may have multiple possible "branches" in their instructions. Consider the following:

Algorithm A.5. On input x, either output 0 or 1.

We say that algorithms of this sort are *nondeterministic*; in contrast, an algorithm is *deterministic* if its instructions do not include such choices. In particular, we say that an algorithm $\mathcal{A}$ *deterministically computes* a function f if it is deterministic and, for any input

$x \in \Sigma^*$, $\mathcal{A}$ outputs the value $f(x) \in \Sigma^*$. In contrast, $\mathcal{A}$ *nondeterministically computes f* if, for any input x, there exists a particular choice of branches such that $\mathcal{A}$ outputs $f(x)$. Thus Algorithm A.5 nondeterministically computes both the functions $x \mapsto 0$ and $x \mapsto 1$. We sometimes view nondeterministic algorithms as computing functions into the power set of Σ^*, so that Algorithm A.5 computes the function $x \mapsto \{0, 1\}$, and we similarly sometimes write $\mathcal{A}(x) = \{0, 1\}$.

An important middle ground is *probabilistic* algorithms. Again, there are many possible models, but the basic idea is that a probabilistic algorithm has access to some source of randomness—say, an arbitrarily long string of independent and uniform coin tosses—which it can use to choose between branches. In this case, it is not enough for there to be some branch which computes a specific function. Instead, we say that an algorithm *computes f with bounded probability* if for any input x,

$$\Pr[\mathcal{A}(x) = f(x)] > \frac{2}{3},$$

where the probability is taken over the randomness of $\mathcal{A}$[1]. In this case, we often think of $\mathcal{A}(x)$ as a probability distribution on Σ^*.

Instead of thinking of algorithms operating directly on binary strings, we usually think of them as operating on encodings of mathematical objects. For example:

Algorithm A.6. On input x a natural number, output the number $2x$.

We say that Algorithm A.6 deterministically computes $x \mapsto 2x$, even though it technically operates on encodings of naturals. While there are many possible encodings, we assume that a reasonable encoding is chosen, so that for instance numbers are encoded in binary, rather than unary. Such details will not be relevant for us.

One more subtlety is important. In general, we require that the description of any algorithm $\mathcal{A}$ is finite. However, we may also consider *non-uniform* algorithms, which are sequences of algorithms $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots)$ such that, on an input of length n, $\mathcal{A}$ delegates to $\mathcal{A}_n$. Non-uniform computation is generally stronger than uniform computation, as non-uniform algorithms may encode nonfinite information, as long as they only use finitely much of this information for each input length and hence for each computation[2].

A.3 Complexity Theory

Each algorithm has an associated *running time*, which is informally the number of steps the algorithm takes on a given input. In particular, for an algorithm $\mathcal{A}$, we say that its running time is the function $T_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ which takes any natural number n to the maximum number of steps $\mathcal{A}$ takes to terminate on any input of length n. Of course, this

[1] The choice of $\frac{2}{3}$ is not particularly important here—generally any constant $c > \frac{1}{2}$ works.

[2] For instance, non-uniform algorithms may solve the halting problem (which asks whether an input algorithm $\mathcal{M}$ eventually terminates), which is uniformly undecidable. In particular, since there are only finitely many Turing machines of a given size, a non-uniform algorithm may simply encode in $\mathcal{A}_n$ the answer to the halting problem for each Turing machine of length n.

notion is not yet precise, as we don't know what a "step" is, but it is easy to make precise in any standard model of computation.

In general, the running time may depend on the formal model of computation in which the algorithm is constructed, but the *complexity-theoretic Church-Turing thesis* states that "reasonable" models of classical computation recover the same inhabitants of sufficiently robust complexity classes, in particular of those we are about to define. This hypothesis is a heuristic, but has been born out in practice.

Definition A.7 (polynomial-time; P, NP)**.** An algorithm $\mathcal{A}$ is *polynomial-time* if $T_{\mathcal{A}} = O(n^k)$ for some constant k. The class P consists of all functions which are deterministically computable by polynomial-time algorithms. The class NP consists of all functions which are nondeterministically computable by polynomial-time algorithms[3].

The general idea is that polynomial-time algorithms are "efficient in practice." It may sometimes occur that the constant factors or the exponent are so large as to render the algorithm practically useless, but in most cases functions in P are efficiently solvable for practical applications, including cryptography. We can now state the most important open problem in computer science:

Conjecture A.8. *We have that P ≠ NP.*

While a proof seems completely out of reach, this conjecture is widely believed, and as we will see is necessary for all of modern cryptography; we will assume it here. An introduction to the modern state of P vs. NP is [And17].

Formalizing probabilistic complexity classes is slightly more subtle. Consider the following case:

Algorithm A.9. On input x, output 1 with probability $1 - 2^{-|x|}$; otherwise count from 0 to $2^{|x|}$ and then output 1.

While this algorithm is almost always polynomial-time, it is not polynomial-time when it takes the second branch. The point is that for probabilistic algorithms, $T_{\mathcal{A}}(n)$ is a probability distribution, not just a fixed number. For our purposes, we require that the algorithm *always* runs in polynomial time. As such:

Definition A.10 (probabilistic polynomial-time; BPP)**.** A probabilistic algorithm is *probabilistic polynomial-time* if, for any choice of random bits, $T_{\mathcal{A}} = O(n^k)$ for some constant k. The class BPP consists of all functions which are computable with bounded probability by a probabilistic polynomial-time algorithm.

For non-uniform algorithms, the situation is also slightly more complicated. In particular, it is too much to allow the machines to be arbitrarily large, as they could simply encode lookup tables for every possible input. As such, we ask that the size of each machine is polynomially bounded.

[3] In fact, we have defined here the classes FP and FNP of polynomially- and nondeterministically-polynomially-computable *function problems*. Formally, P and NP are classes of *decision problems*, which are just subsets L of Σ^*—the algorithm must output 1 if its input is in L, and 0 otherwise. Function and decision problems are extremely closely related—for instance, P = NP if and only if FP = FNP—and we will not distinguish between them here.

Definition A.11 (non-uniform polynomial-time; P/poly)**.** A non-uniform algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \dots)$ is *polynomial-time* if $T_{\mathcal{A}} = O(n^k)$ for some constant k and the size of each $\mathcal{A}_n$ is $O(n^k)$ for some constant k independent of n. The class P/poly consists of all functions which are computable by non-uniform polynomial-time algorithms.

Non-uniform probabilistic algorithms are similarly defined.

Theorem A.12 (Adelman's theorem)**.** *We have that BPP $\subseteq$ P/poly.*

Bibliography

[And17] Scott Anderson. "P $\overset{?}{=}$ NP". In: (2017). URL: https://www.scottaaronson.com/papers/pnp.pdf.

[BK22] Anne Broadbent and Martti Karvonen. "Categorical composable cryptography". In: *Foundations of software science and computation structures*. Vol. 13242. Lecture Notes in Comput. Sci. Springer, Cham, 2022, pp. 161–183. ISBN: 9783030992538. DOI: 10.1007/978-3-030-99253-8_9. URL: https://doi.org/10.1007/978-3-030-99253-8_9.

[BP17] John C. Baez and Blake S. Pollard. "A compositional framework for reaction networks". In: *Reviews in Mathematical Physics* 29.09 (Sept. 2017), p. 1750028. DOI: 10.1142/s0129055x17500283. URL: https://doi.org/10.1142%2Fs0129055x17500283.

[BW90] Michael Barr and Charles Wells. *Category theory for computing science*. USA: Prentice-Hall, Inc., 1990. ISBN: 0131204866.

[Can00] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Paper 2000/067. https://eprint.iacr.org/2000/067. 2000. URL: https://eprint.iacr.org/2000/067.

[Can08] Ran Canetti. *Lecture 11*. Spring 2008. URL: https://www.cs.tau.ac.il/~canetti/f08-materials/scribe11.pdf.

[Can20] Ran Canetti. "Universally Composable Security". In: *J. ACM* 67.5 (Sept. 2020). ISSN: 0004-5411. DOI: 10.1145/3402457. URL: https://doi.org/10.1145/3402457.

[CF01] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 19–40. ISBN: 978-3-540-44647-7.

[CFS16] Bob Coecke, Tobias Fritz, and Robert W. Spekkens. "A mathematical theory of resources". In: *Information and Computation* 250 (2016). Quantum Physics and Logic, pp. 59–86. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2016.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0890540116000353.

[GK96] Oded Goldreich and Hugo Krawczyk. "On the Composition of Zero-Knowledge Proof Systems". In: *SIAM Journal on Computing* 25.1 (1996), pp. 169–192. DOI: 10.1137/S0097539791220688. eprint: https://doi.org/10.1137/S0097539791220688. URL: https://doi.org/10.1137/S0097539791220688.

[GL89] O. Goldreich and L. A. Levin. "A Hard-Core Predicate for All One-Way Functions". In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, pp. 25–32. ISBN: 0897913078. DOI: 10.1145/73007.73010. URL: https://doi.org/10.1145/73007.73010.

[Gog+73] J. A. Goguen et al. *A Junction between Computer Science and Category Theory*. Tech. rep. 1973. URL: https://dominoweb.draco.res.ibm.com/49eae98dc5a21de0852574ff005001c8.html.

[Gol01] O. Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2001. ISBN: 9780521791724.

[HM03] Dennis Hofheinz and Jörn Müller-Quade. *A Paradox of Quantum Universal Composability*. 2003. URL: https://www.quiprocone.org/Hot%20Topics%20posters/muellerquade_poster.pdf.

[Imp95] R. Impagliazzo. "A personal view of average-case complexity". In: *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*. 1995, pp. 134–147. DOI: 10.1109/SCT.1995.514853.

[Jac99] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2014. ISBN: 9781466570269.

[Lin17] Yehuda Lindell. "How to Simulate It—A Tutorial on the Simulation Proof Technique". In: *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*. Springer International Publishing, 2017, pp. 277–346. ISBN: 9783319570488. DOI: 10.1007/978-3-319-57048-8_6.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Vol. 5. New York: Springer-Verlag, 1971, pp. ix+262.

[MM90] José Meseguer and Ugo Montanari. "Petri nets are monoids". In: *Information and Computation* 88.2 (1990), pp. 105–155. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(90)90013-8. URL: https://www.sciencedirect.com/science/article/pii/0890540190900138.

[MR92] Silvio Micali and Phillip Rogaway. "Secure Computation". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 392–404. ISBN: 978-3-540-46766-3.

[Ped91] Torben Pryds Pedersen. "Non-interactive and information-theoretic secure verifiable secret sharing". In: *Annual international cryptology conference*. Springer. 1991, pp. 129–140.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, Aug. 1991. ISBN: 9780262288460. DOI: 10.7551/mitpress/1524.001.0001. URL: https://doi.org/10.7551/mitpress/1524.001.0001.

[PS10] Raphael Pass and Abhi Shelat. *A Course in Cryptography*. 2010. URL: https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[Rie17] Emily Riehl. *Category theory in context*. en. Courier Dover Publications, Mar. 2017.

[Ros21] Mike Rosulek. *The Joy of Cryptography*. 2021. URL: https://joyofcryptography.com.

[Sel11] P. Selinger. "A Survey of Graphical Languages for Monoidal Categories". In: *New Structures for Physics*. Ed. by Bob Coecke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 289–355. ISBN: 978-3-642-12821-9. DOI: 10.1007/978-3-642-12821-9_4. URL: https://doi.org/10.1007/978-3-642-12821-9_4.

[Sim11] Harold Simmons. *An Introduction to Category Theory*. USA: Cambridge University Press, 2011. ISBN: 0521283043.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN: 113318779X.

[SS02] Ahmad-Reza Sadeghi and Michael Steiner. *Assumptions Related to Discrete Logarithms: Why Subtleties Make a Real Difference*. Cryptology ePrint Archive, Paper 2002/126. https://eprint.iacr.org/2002/126. 2002. URL: https://eprint.iacr.org/2002/126.

[Tre09] Luca Trevisan. *Notes for Lecture 27*. Apr. 2009. URL: https://theory.stanford.edu/~trevisan/cs276/lecture27.pdf.

[Unr10] Dominique Unruh. "Universally Composable Quantum Multi-Party Computation". In: *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT'10. French Riviera, France: Springer-Verlag, 2010, pp. 486–505. ISBN: 3642131891. DOI: 10.1007/978-3-642-13190-5_25. URL: https://doi.org/10.1007/978-3-642-13190-5_25.

[Yao82] Andrew C. Yao. "Theory and application of trapdoor functions". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 1982, pp. 80–91. DOI: 10.1109/SFCS.1982.45.