Chapter 1

Introduction

The bird's-eye view of computability: what does it mean, how does it work, where did it come from?

1.1. Approach

If I can program my computer to execute a function, to take any input and give me the correct output, then that function should certainly be called computable. That set of functions is not very large, though; I am not a particularly skilled programmer and my computer is nothing spectacular. I might expand my definition to say if a wizardly programmer can program the best computer there is to execute a function, that function is computable. However, programming languages and environments improve with time, making it easier to program complicated functions, and computers increase in processing speed and amount of working memory.

The idea of "computable" as "programmable" provides excellent intuition, but the precise mathematical definition needs to be an idealized version that does not change with hardware advances. It should capture all of the functions that may be automated even in theory.

To that end, computability theory removes any restriction on time and memory except that a successful computation must use only finitely much of each. When that change is made, the rest of the

details are fairly flexible: with sufficient memory, even the simple language of a programmable calculator is sufficient to implement all computable functions. An explicit definition is in §3.2.

Most computable functions, by this definition, are completely infeasible today, requiring centuries to run and more memory than currently exists on the planet. Why should we include them? One reason is that if we require feasibility, we have to draw a line between feasible and infeasible, and it is unclear where we ought to draw that line. A related reason is that if we do not require feasibility, and then we prove a function is noncomputable, the result is as strong as possible: no matter how much computing technology improves, the function can never be automated. A major historical example of such a proof is described in the next section, and more are in §4.5, from a variety of mathematical fields. These unsolvable problems are often simple statements about basic properties, so intuition may suggest they should be easy to compute.

A rigorous and general definition of computability also allows us to pin down concepts that are *necessarily* noncomputable. One example is randomness, which will be explored in more depth in §9.2. If a fair coin is flipped repeatedly, we do not expect to be able to win a lot of money betting on the outcome of each flip in order. We may get lucky sometimes, but in the long run whatever betting strategy we use should be essentially equivalent in success to betting heads every time: a fifty percent hit rate. That is an intuitive notion of what it means for the flip outcomes to be random. To turn it into mathematics we might represent betting strategies as functions, and say that a sequence of flips is random if no betting strategy is significantly better than choosing heads every time. However, for any given sequence of coin flips C, there is a function that bets correctly on every flip of C. If the class of functions considered to be betting strategies is unrestricted, no sequences will be random, but intuition also says *most* sequences will be random. One restriction we can choose is to require that the betting function be computable; this ties into the idea that the computable functions are those to which we have some kind of "access." If we could never hope to implement a function, it is difficult to argue that it allows us to predict coin flips.

I would like to highlight a bit of the mindset of computability theory here. Computability theorists pay close attention to *uniformity*. A nonuniform proof of computability proves some program exists to compute the function, and typically has some ad hoc component. A uniform proof explicitly builds such a program. A common example is showing explicitly that there is a program P that computes $f(n)$ provided $n \geq N$ for some fixed, finite N, and considering f computable despite the fact that P may completely fail at finding $f(n)$ for $0 \leq n < N$ (in computability theory, functions are on the natural numbers; see §3.4 for why that is not as stringent a restriction as it may seem). P is sufficient because for any given sequence of N numbers, there is a program Q that assigns those numbers in order as $f(0)$ through $f(N-1)$, and uses P to find the output for larger numbers. The function we want to compute must have *some* fixed sequence of outputs for the first N inputs, and hence via the appropriate Q it may be computed. It is computable, although we would need to magically (that is, nonuniformly) know the first N outputs to have the full program explicitly. Nonuniformity in isolation is not a problem, but it reduces the possible uses of the result. We will discuss uniformity from time to time as more examples come up.

Computability theorists more often work in the realm of the noncomputable than the computable, via approximations and partial computations. Programs that do not always give an output are seen regularly; on certain inputs such a program may just chug and chug and never finish. Of course, the program either gives an output or doesn't, and in many areas of mathematics we would be able to say "if the program halts, use the output in this way; if not, do this other computation." In fact we *could* do that in computability theory, but the question of whether any given program halts is a noncomputable one (see §4.1). Typically we want to complete our constructions using as little computational power as we can get away with, because that allows us to make stronger statements about the computability or noncomputability of the function we have built.

To succeed in such an environment, the construction must continue while computations are unfinished, working with incomplete and possibly incorrect assumptions. Mistakes will be made and need

to be repaired. The standard means of dealing with this situation is a *priority argument* (§6.2), which breaks the goals of the construction into small pieces and orders them. A piece (*requirement*) that is earlier in the ordering has higher priority and gets to do what appears at the moment to satisfy its goal even if that is harmful to later requirements. When done correctly, each requirement can be met at a finite stage and cease harming the lower-priority requirements, and each requirement can recover from the damage it sustains from the finitely many higher-priority requirements. Satisfaction of requirements cascades irregularly from the beginning of the order down.

1.2. Some History

The more early work in computability theory you read, the more it seems its founding was an inevitability. There was a push to make mathematics formal and rigorous, and find mechanical methods to solve problems and determine truth or falsehood of statements; hence there was a lot of thought on the nature of mechanical methods and formality. For more on this early work, I recommend John Hopcroft's article "Turing Machines" [41] and two survey papers by Martin Davis [20, 21]. If you wish to go deeper philosophically (and broader mathematically), try van Heijenoort [86] and Webb [88].

David Hilbert gave an address in 1900 in which he listed problems he thought should direct mathematical effort as the new century began [37]. His tenth problem, paraphrased, was to find a procedure to determine whether any given multivariable polynomial equation with integer coefficients (a *Diophantine* equation) has an integer solution. Hilbert asked for "a process according to which it can be determined by a finite number of operations" whether such a solution exists. For the specific case of single-variable Diophantine equations, mathematicians already had the rational root test; for an equation of the form $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0 = 0$, any rational solution must be of the form $\pm\frac{r}{s}$ where r divides a_0 and s divides a_n. One may manually check each such fraction and determine whether any of them yields equality.

In 1910, the first volume of Alfred North Whitehead and Bertrand Russell's *Principia Mathematica* was published [89]. This ultimately

three-volume work was an effort to develop all mathematics from a small common set of axioms, with full rigor at every step. Russell and Whitehead wanted to remove vagueness and paradox from mathematics; every step in their system could be checked mechanically.

It seems this might take all creativity out of mathematics, as all possible theorems would eventually be produced by the mechanical application of logical deduction rules to axioms and previously generated theorems. However, in 1931 Gödel showed it was impossible for such a system to produce all true mathematical statements [30]. He used the mechanical nature of the system, intended for rigor, and showed it allowed a system of formula encoding that gave access to self-reference: he produced a formula P that says "P has no proof." If P is true, it is unprovable. If the negation of P is true, P has a proof, since that is the assertion made by P's negation. Therefore, unless Russell and Whitehead's system is internally inconsistent, P must be true, and hence unprovable. We will see a proof of Gödel's Incompleteness Theorem via undecidable problems in §5.3.

Principia Mathematica was a grand undertaking and one might expect it to fail to fulfill all its authors' hopes. However, Hilbert's quest was doomed as well. The proof that there can be no procedure to determine whether an arbitrary Diophantine equation has integer roots was not completed until 1973 [61], but in 1936 Church made the first response suggesting that might be the case [14]. He wrote:

> There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, \ldots, x_n) = 2$ is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, \ldots, x_n$ as free variables. [footnote: The selection of the particular positive integer 2 instead of some other is, of course, accidental and non-essential.]
>
> ... The purpose of the present paper is to propose a definition of effective calculability which is thought to correspond satisfactorily to the somewhat vague

> intuitive notion in terms of which problems of this class are often stated, and to show, by means of an example, that not every problem of this class is solvable.

Church's major contribution here is the point that we need some *formal* notion of "finite process" to answer Hilbert. He proposes two options in this paper: the lambda calculus, due to him and Kleene, and recursive functions, defined originally by Gödel [30] (after a suggestion by Herbrand) and modified by Kleene. Shortly thereafter Kleene proposed what we now call the partial recursive functions [44]. It was not widely accepted at the time that any of these definitions was a good characterization of "effectively computable," however. It was not until Turing developed his Turing machine [85], which *was* accepted as a good characterization, and it was proved that Turing-computable functions, lambda-computable functions, and partial recursive functions are the same class, that the functional definitions were accepted. All three of these formalizations of computability are studied in Chapter 3. The idea that not all problems are solvable comes up in Chapter 4, along with many of the tools used in such proofs.

Both Gödel's Incompleteness Theorem and Church's unsolvability result treat the limitations of mechanical, or algorithmic, procedures in mathematics. As is common in mathematics, these new ideas and tools took on a life of their own beyond answering Hilbert or finding the major flaw in Russell and Whitehead's approach to mathematics. The new field became known as recursion theory or computability theory. Chapters 5–8 explore some of the additional topics and fundamental results of the area, and Chapter 9 contains a survey of some areas of current interest to computability theorists.

1.3. Notes on Use of the Text

My intent is that Chapter 2 will be covered on an as-needed basis, and I have tried to include references to it wherever applicable throughout the text. However, the vocabulary in §§2.1 and 2.2 is needed throughout, so they should be read first if unfamiliar. Returning to §1.1 after reading through Chapter 4 may be helpful as well.

The core material is in Chapters 3 through 7. In those, without losing continuity, §§3.7, 4.5, 5.3, 6.2, and 6.3 may be omitted; if §6.2 is omitted, §5.4 may also be.

Chapters 8 and 9 should be covered as interest warrants. There is no interdependence between sections of these chapters except that §§9.4 and 9.5 both draw on §9.3, and §9.1 leans lightly on §8.3.

1.4. Acknowledgements and References

These notes owe a great debt to a small library of logic books. For graduate- and research-level work I regularly refer to *Classical Recursion Theory* by P. G. Odifreddi [68], *Theory of Recursive Functions and Effective Computability* by H. Rogers [75], and *Recursively Enumerable Sets and Degrees* by R. I. Soare [82]. The material in here owes a great deal to those three texts. More recently, I have enjoyed A. Nies' book *Computability and Randomness* [67]. As you will see, M. Davis' *The Undecidable* [18] was a constant presence on my desk as well.

In how to present such material to undergraduates, I was influenced by such books as *Computability and Logic* by Boolos, Burgess, and Jeffrey [10], *Computability* by Cutland [17], *A Mathematical Introduction to Logic* by Enderton [25], *An Introduction to Formal Languages and Automata* by Linz [57], and *A Transition to Advanced Mathematics* by Smith, Eggen, and St. Andre [81].

I have talked to many people about bits and pieces of the book, getting clarifications and opinions on my exposition, but the lion's share of gratitude must go to Denis Hirschfeldt. Many thanks are due also to the students in the three offerings of Computability Theory I gave as I was writing this text, first as course notes and then with my eye to publishing a book. With luck, their questions as they learned the material and their comments on the text have translated into improved exposition.

3.4. Coding and Countability

So far we've computed only with natural numbers. How could we define computation on domains outside of $\mathbb{N}$? If the desired domain is countable, we may be able to encode its members as natural numbers. For example, we could code $\mathbb{Z}$ into $\mathbb{N}$ by using the even natural numbers to represent nonnegative integers, and the odd to represent negative integers. Specifically, we can write the following computable function.

$$f(k) = \begin{cases} 2k & k \geq 0 \\ -2k-1 & k < 0 \end{cases}$$

We might also want a subset of $\mathbb{N}$ to serve in place of all of the natural numbers. For example, we might let every natural number be interpreted as its double. In terms of coding the even numbers E into $\mathbb{N}$, the function is now $g(k) = k/2$.

The important property we need to treat natural numbers as codes of elements from another set S is a bijection between S and $\mathbb{N}$ that is computable with computable inverse. Sets for which such bijections, or *coding functions*, exist are called *effectively countable*. The image of the element of S under this bijection is its *code*; the fact that the function is bijective ensures every natural number codes some unique element of S. Finally, the effectiveness of the function gives us its usefulness: any infinite countable set is in bijection with $\mathbb{N}$, but we can't use that bijection in a Turing machine unless it is computable.

Note that unlike countability in general, effective countability is not guaranteed for subsets of effectively countable sets. One example is in Theorem 3.5.1; more will appear in Chapter 5.

There are two ways to compute on coded input.

(1) The Turing machine can decode the input, perform the computation, and encode the answer.

(2) The Turing machine can compute on the encoded input directly, obtaining the encoded output.

Exercise 3.4.1. Consider $\mathbb{Z}$ encoded into $\mathbb{N}$ by f above. Write a function that takes $f(k)$ as input and outputs $f(2k)$ using approach 2 above. An algebraic expression will suffice.

Coding is often swept under the rug; in research papers one generally sees at most a comment to the effect of "we assume a coding of [our objects] as natural numbers is fixed." It is a vital component of computability theory, however, as it removes the need for separate definitions of *algorithm* for different kinds of objects.

To move into $\mathbb{N}^2$, the set of ordered pairs of natural numbers, there is a standard *pairing function* indicated by angle brackets.

$$\langle x, y \rangle := \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$$

For longer tuples we iterate, so for example $\langle x, y, z \rangle := \langle \langle x, y \rangle, z \rangle$. The pairing function and its iterations show that for every k, $\mathbb{N}^k$ is effectively countable. They also let us treat multivariable functions in the same way as single-input functions.

The pairing function is often given as a magic formula from on high, but it's quite easy to derive. You may be familiar with Cantor's proof that the rational numbers are the same size as the natural numbers, where he walks diagonally through the grid of integer-coordinate points in the first quadrant and skips any that have common factors (if not, see Appendix A.3). We can do essentially that now, though we won't skip anything.

Starting with the origin, we take each diagonal and walk down it from the top (see Figure 3.1). The number of pairs on a given diagonal is one more than the sum of the entries of each pair. The number of pairs above a given (x, y) on its own diagonal is x, so if we want to number pairs from 0, we let (x, y) map to

$$1 + 2 + \ldots + (x + y) + x,$$

where each term except the last corresponds to a diagonal below (x, y)'s diagonal. This sums to

$$\frac{(x + y + 1)(x + y)}{2} + x = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y).$$

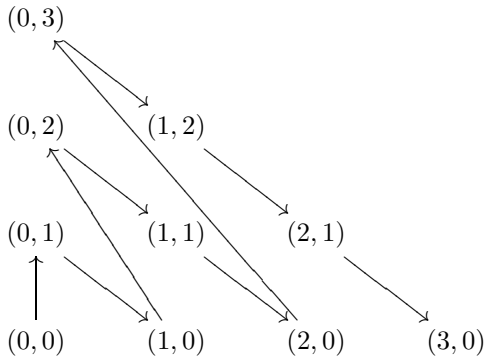

Figure 3.1. Order of pair counting for pairing function.

If you are unfamiliar with the formula for the summation of the integers 1 through n, you can find it in Appendix A.2; it is not needed beyond this point.

A coding function need not have an explicit formula like the polynomial for the pairing function. The rational numbers, $\mathbb{Q}$, may be coded similarly to $\mathbb{N}^2$, using their fractional representation. Let 0 map to 0, and then map the strictly positive numbers into $(\mathbb{N} - \{0\}) \times (\mathbb{N} - \{0\})$. Skipping any pairs (x, y) such that x/y is not in least terms, list pairs in the same order as in the pairing function. The code of a positive rational is the position in the list of its least-terms fractional representation, so 1 maps to 1, $1/2$ maps to 2, and 2 maps to 3, the positions of $(1, 1)$, $(1, 2)$, and $(2, 1)$, respectively. To account for negative rational numbers, double all of these codes, and subtract 1 if the rational is negative; 0 still maps to 0, but 1 maps to 2 (-1 maps to 1) and $1/2$ maps to 4 ($-1/2$ maps to 3).

The important set $\bigcup_{k\geq 0} \mathbb{N}^k$, finite tuples of any length, is also effectively countable (note $\mathbb{N}^0 = \{\emptyset\}$). The function

$$\tau : \bigcup_{k\geq 0} \mathbb{N}^k \to \mathbb{N}$$

given by $\tau(\emptyset) = 0$ and

$$\tau(a_1, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \ldots + 2^{a_1+\ldots+a_k+k-1}$$

demonstrates the effective countability. A singleton – that is, an element of $\mathbb{N}$ itself – is mapped to a number with binary representation using a single 1. An n-tuple maps to a number whose binary representation uses exactly n 1s.

Exercise 3.4.2. (i) Find the images under τ of the tuples $(0,0)$, $(0,0,0)$, $(0,1,2)$, and $(2,1,0)$.

(ii) What is the purpose of summing subsequences of a_i and adding $1, 2, \ldots, k-1$ in the exponents? What tuples get confused if you perform only one of those actions?

(iii) Prove that τ is a bijection.

Exercise 3.4.3. Let A and B be effectively countable, infinite sets.

(i) If A and B are disjoint, prove that $A \cup B$ is effectively countable.

(ii) If A and B are not necessarily disjoint, prove that $A \cup B$ is effectively countable.

(iii) Prove that $A \cap B$ is effectively countable or finite.

Exercise 3.4.4. (i) Show that if a class A of objects is constructed recursively using a finite set of basic objects and a finite collection of computable closure operators (see §2.5), A is effectively countable.

(ii) Show that even if the sets of basic objects and rules in part (i) are infinite, as long as they are effectively countable, A is also effectively countable. §5.1 may be helpful.

Example 3.4.5. This example is vital for the rest of the text. The set of Turing machines is, in fact, effectively countable; the TMs may be coded as natural numbers. One way to code them is to first encode individual quadruples as single numbers, so the machine is represented by a finite set, and then encode finite subsets of $\mathbb{N}$ as single numbers, as in Exercise 3.1.4. Similarly to the rational numbers, we can encode a subset of the quadruples and then spread them out to "fit in" the rest. The symbols that may appear in the middle two slots of a quadruple are drawn from a fixed finite list, but the indices of the states that appear first and last may be arbitrarily large. Therefore, we start by encoding $\langle q_i, \cdot, \cdot, q_j \rangle$ as $\langle i, j \rangle$ by pairing, and then multiply those values by a large enough number to fit exactly the possibilities

for the middle two slots. That value will depend on what symbols are allowed; if the symbol set is only $*$ and 1, it will be 8: there are 8 pairs with first element $*$ or 1 and second element $*$, 1, L, or R. Order the pairs in some fixed way; perhaps all pairs beginning with $*$ precede all pairs beginning with 1 and within those blocks they are ordered according to the list of second elements above. In that case, $\langle q_3, *, L, q_3\rangle$ maps to $8\langle 3, 3\rangle + 2$ and $\langle q_3, 1, R, q_3\rangle$ to $8\langle 3, 3\rangle + 7$; $\langle q_0, *, *, q_1\rangle$ maps to $8\langle 0, 1\rangle + 0$.

Conversely, the natural number n will be read as $k + \ell$, where $0 \leq \ell \leq 7$ and k is a multiple of 8. The quotient $k/8$ is decoded into $\langle i, j\rangle$ and ℓ is used to find the symbol read and action taken.

It is important to note that these codes will include "junk machines," codes that may be interpreted as TMs but which give machines that don't do anything. There will also be codes that give machines that, while different, compute the same function – they are different implementations of the function. In fact, we can prove the Padding Lemma, Exercise 3.4.7, after a bit of vocabulary.

Definition 3.4.6. We call the code of a Turing machine its *index*, and say when we choose a particular coding that we *fix an enumeration* of the Turing machines (or, equivalently, the partial recursive functions). It is common to use φ for partial recursive functions; φ_e is the e^{th} machine/function in the enumeration, the machine with index e, and the machine that encodes to e.

Indices are often called *Gödel numbers*, because Gödel introduced the notion of coding formulas in order to prove incompleteness [30]. Coding formulas and sequences of formulas (such as proofs) as numbers allows the statement "there is no proof of me" to be expressed as, roughly, "no number codes a proof of the formula coded by this number," where the number in question at the end codes the given formula itself.

Exercise 3.4.7. (The Padding Lemma). Prove that given any index of a Turing machine M, there is a larger index which codes a machine that computes the same function as M.

We often use the indices simply as tags, to put an ordering on the functions, but it is often important to remember that the index *is* the function, in a very literal way. Given the index, it is primitive recursive to obtain the entire machine. Conversely, you will also see indices defined by description of the function of which they code an implementation. For example, letting e be such that

$$\varphi_e(x,y) = \begin{cases} x+y & y < 2 \\ xy & 2 \leq y \leq 4 \\ x^y & y \geq 4 \end{cases}$$

is a perfectly good definition of e. In principle, we can write that function as a Turing machine, compute the machine's index, and use that value as e.

In fact, we can define functions in this way as well. An example of such a definition is "let $f(x)$ index the function with domain $\{x\}$." Given x, we can create a Turing machine that halts on x alone, and encode that as an index. That is the value we would like to call $f(x)$. The missing piece is *uniformity*: to claim f is a computable function, the process of creating the Turing machine from x must be independent of the value of x. That is, while it will create different machines from different values of x, there is one instruction sheet we can give to the person writing the machines that covers all the possibilities. Here, that is straightforward: have, say, $x+1$ states, x of which count 1s from 0 to x and halt if there are the right number, the last of which is the "nonhalting" state that is entered when the 1s run out too early or last too long. The value $f(x)$ is the index of the machine for x.

Exercise 3.4.8. Explicitly give a template for a Turing machine with domain $\{x\}$, with x a natural number. You are not required to have exactly $x+1$ states.

Another collection of objects commonly indexed is the finite sets, as in Exercise 3.1.4. The n^{th} finite set, or set corresponding to n in the bijection, is typically denoted D_n.

The key points of this section for our later work are the following:

(1) Via coding, we can treat any effectively countable set as though it were $\mathbb{N}$.

(2) A single natural number may mean a great variety of objects; Turing machines are set to interpret their input in a specific way.

(3) We can fix an enumeration of the Turing machines (equivalently, the partial computable functions); the index of a particular machine will *be* that machine in code. It is understood that the coding is fixed from the start so we are never trying to decode with the wrong bijection.

3.5. A Universal Turing Machine

From the enumeration of all Turing machines and the pairing function we can define a *universal* Turing machine U; that is, a machine that will emulate *every* other machine. Simply set

$$U(\langle e, x\rangle) = \varphi_e(x).$$

U decodes the single number it receives into a pair (e, x), decodes e into the appropriate set of quadruples, and uses x as input, acting according to the quadruples it decoded. This procedure is computable because decoding the pairing function, decoding Turing machines from indices, and executing quadruples on a given input are computable procedures. Turing [85] gives an explicit, full construction of a universal machine.

Note that of course there are infinitely many universal Turing machines, as there are for any program via padding, and that a universal machine exists for any collection of functions that may be indexed. Although we will use U and analogously defined universal machines for other indexings in this section, typically we simply refer to the individual indexed functions, using $\varphi_e(x)$ instead of $U(\langle e, x\rangle)$.

We are now in the position to demonstrate a very practical reason to allow partial functions in our definition of computability. Recall that by *total computable function* we mean a function from the class of partial computable functions which happens to be total. The following theorem also gives an example of a subset of the effectively countable set $\mathbb{N}$ that is not itself effectively countable: the indices of the total computable functions.

Theorem 3.5.1. *There is no computable indexing of the total computable functions.*

Proof. Suppose the contrary and let f_e denote the e^{th} function in an effective enumeration of all total computable functions. Let $u(\langle x, y\rangle)$ be a universal machine for the total computable functions; it is itself computable, and since $f_x(y)$ is defined for all x and y, u is also total. We define a new function as follows.

$$g(e) = u(\langle e, e\rangle) + 1$$

It is clear that g is total computable, since it is successor composed with the total computable function u. Hence g must have an index; that is, there must be some e' such that $g = f_{e'}$. However, $g(e') = u(\langle e', e'\rangle) + 1 = f_{e'}(e') + 1 \neq f_{e'}(e')$, which is a contradiction. Therefore no such indexing can exist. □

As an aside, those who are familiar with Cantor's proof that the real numbers are uncountable will notice a distinct similarity (if not, see Appendix A.3). This is an example of a *diagonal argument*, where you accomplish something with respect to the e^{th} Turing machine using e. Of course you need not use literally e, as we will see in later chapters.

We have two choices, then, with regard to the collection of functions we call "computable:" to have them all be total, but fail to have an indexing of them, or to include partial functions and be able to enumerate them. We will see many proofs that rely completely on the existence of an indexing in order to work; this combined with the justifications in §3.3.3 weigh heavily on the side of allowing partial functions to be called computable.

We now return to the primitive recursive functions, and a way to break out of that class other than the Ackermann function.

Theorem 3.5.2. *There is a function f such that $f(\langle n, x\rangle)$ computes the n^{th} primitive recursive function on input x.*

Sketch of Proof. We may code the primitive recursive functions by their derivation from the basic functions through the application of closure operators. The basic functions and closure operators are given

values, which are then combined via pairing in ways that indicate how closure operators are being applied to basic functions and functions appearing earlier in the derivation. This may be done computably, giving an indexing of the primitive recursive functions; call the n^{th} such function θ_n. Define $f(\langle n, x\rangle) = \theta_n(x)$; f operates analogously to U, as described at the beginning of the section. □

Corollary 3.5.3. *There is a function that is total recursive but not primitive recursive.*

Proof. From f as in Theorem 3.5.2, define $g(n) = f(\langle n, n\rangle)+1$. Since all primitive recursive functions are total and codings are bijective, f is total, and hence g is total. However, g is not primitive recursive, because for every n, it differs from the n^{th} primitive recursive function on input n. □

Notice that this is essentially identical to Theorem 3.5.1, but in a different setting and hence with a different conclusion.

Corollary 3.5.4. *The universal function defined in Theorem 3.5.2 is not primitive recursive.*

Proof. The function g from the proof of Corollary 3.5.3 may be written rigorously as

$$g(n) = S(f(\langle P_1^1(n), P_1^1(n)\rangle)),$$

which is a valid primitive recursive derivation, with three applications of composition, provided all the component functions are primitive recursive. Since g is not primitive recursive, one of the component functions must fail to be. As successor, the pairing function, and projection are all primitive recursive, f must be the weak link. □

Exercise 3.5.5. A function h *dominates* a function g if there is some $N \in \mathbb{N}$ such that $h(n) \geq g(n)$ for all $n \geq N$.

(i) Construct a function h that dominates all the primitive recursive functions $\{\theta_e\}_{e\in\mathbb{N}}$ (it is permitted that different θ_e require distinct values of N), and prove that it does. There are at least three natural ways to define h.

(ii) Is h primitive recursive? Prove or refute.

For example, suppose we want to find an index for $\varphi_x + \varphi_y$ uniformly in x and y. We can let $f(x, y, z) = \varphi_x(z) + \varphi_y(z)$ by letting it equal $U(\langle x, z\rangle) + U(\langle y, z\rangle)$, so everything that was either in input or index is now in input. Then the *s-m-n* theorem gives us a computable function $s(x, y)$ such that $\varphi_{s(x,y)}(z) = f(x, y, z)$ (f is φ_i some fixed i; $s(x, y) = s^1_2(i, x, y)$), so $s(x, y)$ is an index for $\varphi_x + \varphi_y$ as a (total computable) function of x and y.

Exercise 4.3.4. Find an index for the composition $\varphi_x \circ \varphi_y$ uniformly in x and y.

4.4. The Recursion Theorem

Kleene's recursion theorem, though provable in only a few lines, is probably the most conceptually challenging theorem in fundamental computability theory. It is extremely useful – vital, in fact – for a large number of proofs in the field. We will discuss this after meeting the theorem and some of its corollaries.

Recall that equality for partial functions is the assertion that when one diverges, so does the other, and when they converge it is to the same output value.

Theorem 4.4.1 (Recursion or Fixed-Point Theorem)**.** *Suppose that f is a total computable function. Then there is a number n such that $\varphi_n = \varphi_{f(n)}$. Moreover, n is computable from an index for f.*

Proof. This is the "magical" proof of the theorem, which will be expanded later in the section. By the *s-m-n* theorem there is a total computable function $s(x)$ such that for all x and y

$$\varphi_{f(\varphi_x(x))}(y) = \varphi_{s(x)}(y).$$

Let m be any index such that φ_m computes the function s; note that s and hence m are computable from an index for f. Rewriting the statement above yields

$$\varphi_{f(\varphi_x(x))}(y) = \varphi_{\varphi_m(x)}(y).$$

Then, setting $x = m$ and letting $n = \varphi_m(m)$ (which is defined because s is total), we have

$$\varphi_{f(n)}(y) = \varphi_n(y)$$

as required. □

The recursion theorem gives a fixed point at the index level. It need not be the case that $n = f(n)$ – in fact generally that will be false – but n and $f(n)$ are codes for implementations of the same function.

Corollary 4.4.2. *There is some n such that $\varphi_n = \varphi_{n+1}$.*

Corollary 4.4.3. *If f is a total computable function, then there are arbitrarily large numbers n such that $\varphi_{f(n)} = \varphi_n$.*

Corollary 4.4.4. *If $f(x, y)$ is any partial computable function, there is an index e such that $\varphi_e(y) = f(e, y)$.*

Exercise 4.4.5. (i) Prove Corollary 4.4.3. Note that we might obtain a fixed point via another function suitably related to f.

(ii) Prove Corollary 4.4.4. It requires both the recursion theorem and the *s-m-n* theorem.

Exercise 4.4.6. Prove the following applications of Corollary 4.4.4:

(i) $(\exists n)(\varphi_n(x) = x + n)$

(ii) $(\exists n)(\varphi_n(x) = n)$

(iii) $(\exists n)(\operatorname{dom} \varphi_n = \{n\})$

(iv) $(\exists n)(\operatorname{dom} \varphi_n = \operatorname{rng} \varphi_n = \{n\})$

Exercise 4.4.7. Show that there exists some n such that $\varphi_n(x, y) = xn^y$.

The recursion theorem allows us to prove that a large collection of functions are noncomputable. The set $A \subseteq \mathbb{N}$ is an *index set* if it has the property that if $x \in A$ and $\varphi_x = \varphi_y$, then $y \in A$.

Theorem 4.4.8 (Rice's Theorem)**.** *Suppose that A is an index set not equal to $\emptyset$ or $\mathbb{N}$. Then χ_A is not computable.*

Proof. We work by contradiction, supposing χ_A is computable. Set some $a \in A$ and $b \notin A$ and consider the following function.

$$f(x) = \begin{cases} a & \chi_A(x) = 0 \\ b & \chi_A(x) = 1 \end{cases}$$

Apply the recursion theorem. □

Exercise 4.4.9. Complete the proof of Rice's theorem.

Rice's theorem is a strong statement about our inability to pluck out particular partial computable functions. There is no individual function f for which it is computable to decide whether a given index gives an implementation of f, and in fact there is *no* collection of functions save the empty set and the collection of all functions for which the indices can be distinguished computably.

The recursion theorem also gives results about enumeration of Turing machines. In particular, the least index of each function cannot be listed in order.

Theorem 4.4.10. *Suppose that f is a total increasing function such that*

(i) *if $m \neq n$, then $\varphi_{f(m)} \neq \varphi_{f(n)}$.*

(ii) *$f(n)$ is the least index of the function $\varphi_{f(n)}$.*

Then f is not computable.

Proof. Suppose f satisfies the conditions of the theorem. By (i), f cannot be the identity, so since it is increasing there is some k such that for all $n \geq k$, $f(n) > n$. Therefore by (ii), $\varphi_{f(n)} \neq \varphi_n$ for every $n \geq k$. However, if f is computable, this violates Corollary 4.4.3. □

We now expand the proof of the recursion theorem. One can view an index-level fixed point (n such that $\varphi_n = \varphi_{f(n)}$) as what can be salvaged from a failed attempt at a literal fixed point (n such that $n = f(n)$).

Let $\delta(e)$ be the diagonal function $\varphi_e(e)$. For any partial computable function f, there is some $\hat{e}$ such that $f \circ \delta = \varphi_{\hat{e}}$ (infinitely many such $\hat{e}$, in fact). By definition of δ, $\delta(\hat{e})$ and $f \circ \delta(\hat{e})$ must be equal. However, that gives a literal fixed point for f, and functions certainly exist that have no literal fixed points. In such a case, δ must diverge on $\hat{e}$ (since the theorem assumes f is total), and we do not get our fixed point.

However, if we loosen our requirement to index-level fixed points, we can be a little more subtle, using the *s-m-n* theorem to define a

total function that is "close enough" to δ. Let e be such that

$$\varphi_e(i,x) = \begin{cases} \varphi_{\varphi_i(i)}(x) & \varphi_i(i)\downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

We could be more concise and say $\varphi_e(i,x) = \varphi_{\varphi_i(i)}(x)$, with the understanding that divergence of the index is divergence of the whole computation. By the *s-m-n* theorem, this is $\varphi_{S_1^1(e,i)}(x)$; let $d(i) = S_1^1(e,i)$ for this e.

Whenever $\delta(i)\downarrow$ (i.e., $\varphi_i(i)\downarrow$), $d(i)$ and $\delta(i)$ index implementations of the same function. However, unlike δ, d is total: when $\delta(i)\uparrow$, $\varphi_{d(i)}(x)\uparrow$ for all x, but $d(i)$ itself is defined.

Let us repeat our original attempt at a fixed point with d in place of δ, now letting $\hat{e}$ be such that $\varphi_{\hat{e}} = f \circ d$. Consider the result of applying $f \circ d$ to $\hat{e}$.

$$\varphi_{f\circ d(\hat{e})}(x) = \varphi_{\varphi_{\hat{e}}(\hat{e})}(x) = \varphi_{\delta(\hat{e})}(x) = \varphi_{d(\hat{e})}(x)$$

The first equality is by choice of $\hat{e}$ and the second is by definition of δ. Notice that since $f \circ d$ and hence $\varphi_{\hat{e}}$ are total, $\delta(\hat{e})$ must converge. The definition of d gives the final equality. We may abbreviate and rewrite a bit to see we have shown

$$\varphi_{f(d(\hat{e}))}(x) = \varphi_{d(\hat{e})}(x),$$

or in other words, that $d(\hat{e})$ is an index-level fixed point for f.

For the theorem's "moreover," note that d was defined independently from f, so with an index for f we can find an index for $f \circ d$ and hence an index-level fixed point for f.

This is extraordinarily useful in constructions. Many of its uses can be summed up as building a Turing machine using the index of the finished machine. The construction will have a line early on like "We construct a partial computable function ψ and assume by the recursion theorem that we have an index e for ψ." This looks insane, but it is completely valid. The construction, which will be computable, is the function for which we seek a fixed point (at the index level). Computability theorists think of a construction as a program. It might have outside components – the statement of the theorem could say "For every function f of this type, ..." – and the construction's if/then statements will give different results depending

on which particular f was in play, but such variations will be *uniform*, as described in §4.3. If we give the construction the input e to be interpreted as the index of a partial computable function, it can use e to produce e', which is an index of the function ψ it is trying to build. The recursion theorem says the construction will have a fixed point, some i such that i and i' both index the same function. Furthermore, this fixed point will be *computable* from an index for the construction itself, which by its uniformity has such a well-defined index. That last detail allows the claim to having the index of ψ from the beginning.

4.5. Unsolvability

The word *solvable* is a synonym of *computable* used in particular contexts. In general, it is used to describe the ability to compute a solution to a problem stated not as a function, but as an algebraic or combinatorial question. *Decidable* is another synonym used in the same contexts as solvable.

We have seen an undecidable problem: the halting problem K in §4.1. In this context the problem would be stated as "is there an algorithm to decide, for any e, whether the e^{th} Turing machine halts on input e?"

The celebrity examples are Diophantine equations and the word problem for groups; the latter will be discussed in §4.5.3. As discussed in §1.2, in 1900 Hilbert posed a list of problems and goals to drive mathematical development [37]. The tenth problem on the list asked for an algorithm to determine whether an arbitrary multivariable polynomial equation $P = 0$, where the coefficients of P are all integers, has a solution in integers. At the time, the idea there *may not be* any such algorithm did not occur. In 1970, after a lot of work by a number of mathematicians, Matiyasevich proved the problem is unsolvable [61]. The full proof and story are laid out in a paper by Davis [19].

The method is to show that every Turing machine may be somehow "encoded" in a Diophantine equation so that the equation has an integer solution if and only if the machine halts. The fact that we cannot always tell whether a Turing machine will halt shows we

It is easiest to imagine this process in terms of Turing machines, which clearly have step-by-step procedures. We run one step of the computation of $f(0)$. If it halts, then we know $f(0)$. Either way, we run two steps of the computation of $f(1)$, and if necessary, two steps of the computation of $f(0)$. Step (or *stage*) n of this procedure is to run the computations of $f(0)$ through $f(n-1)$ each for n steps, minus any we've already seen halt (though since they only add finitely many steps, there's no harm in including them[1]). Since every computation that halts must halt in finitely many stages, each element of f's domain will eventually give its output. Any collection of computations we can index can be dovetailed.

The state of the computation after s steps of computation is denoted with a subscript s: $f_s(n)$. We may use our halting and diverging notation: $f_s(n)\downarrow$ or $f_s(n)\uparrow$. Note that $f_s(n)\uparrow$ does not imply $f(n)\uparrow$; it could be that we simply need to run more steps. Likewise, $f_s(n)\downarrow$ does not mean $f_{s-1}(n)\uparrow$; it means convergence has happened at some step no later than s.

If we are drawing from an indexed list of functions, the stage may share the subscript with the index: $\varphi_{e,s}(n)$. Sometimes the stage number is put into brackets at the end of the function notation, as $\varphi_e(n)[s]$; this will be useful when more than just the function is approximated, as in §6.1. In this case the up or down arrow goes after everything: $\varphi_e(n)[s]\downarrow$.

5.2. Computing and Enumerating

Recall that the characteristic function of a set A (Definition 3.1.1), denoted χ_A or simply A, is the function outputting 1 when the input is a member of A and 0 otherwise. It is total, but not necessarily computable.

Definition 5.2.1. A set is *computable* (or *recursive*) if its characteristic function is computable.

[1]Likewise, we could record our stopping point and just run one more step of each computation plus the first step of an additional computation each time instead of starting from the beginning, but there is no harm in starting over each time. Remember that *computable* does not imply *feasible*. As another side note, this procedure would cause $f(i)$ to have its $(n-i+1)^{st}$ step run at stage n, making the dovetailing truly diagonal.

The word *effective* is often used as a synonym for computable and recursive, but only in the context of procedures (you might say a given construction is effective instead of saying it is recursive or computable; it would be strange to say a set is effective). Note, however, that in the literature they are not always exactly synonymous! Exercise 5.2.21 introduces the notion of *computably inseparable* sets. While *recursively inseparable* is an equivalent term, the *effectively inseparable* sets are a different collection.

A computable characteristic function is simply a computable procedure that will answer "is n in A?" for any n, correctly and in finite time.

Claim 5.2.2. (i) *The complement of a computable set is computable.*

(ii) *Any finite set is computable.*

Proof. (i) Simply note $\chi_{\overline{A}} = 1 - \chi_A$, so the functions are both computable or both noncomputable.

(ii) A finite set may be "hard-coded" into a Turing machine, so the machine has instructions which essentially say "if the input is one of these numbers, output 1; else output 0."

□

Part (ii) is the heart of most nonuniformity. Any finite amount of information is computable, so we may assume it without violating the computability of a procedure. However, to make such an assumption infinitely many times is typically not computable. This is discussed again at the end of §5.4.

Rice's Theorem 4.4.8 gives a large number of noncomputable sets: all nontrivial index sets. Not all noncomputable sets are created equal, of course, and in particular we pluck out sets that are *computably approximable*, in the following sense.

Definition 5.2.3. A set is *computably enumerable* (or *recursively enumerable*, abbreviated as *c.e.* or *r.e.*) if there is a computable procedure to list its elements (possibly out of order and with repeats).

That definition is perhaps a little nebulous. Here are some additional characterizations:

Proposition 5.2.4. *Given a set A, the following are equivalent.*

(i) *A is c.e.*

(ii) *A is the domain of a partial computable function.*

(iii) *A is the range of a partial computable function.*

(iv) *$A = \emptyset$ or A is the range of a total computable function.*

(v) *There is a total computable function $f(x, s)$ such that for every x, $f(x, 0) = 0$, there is at most one s such that $f(x, s + 1) \neq f(x, s)$, and $\lim_s f(x, s) = \chi_A(x)$.*

(vi) *There is a computable sequence of finite sets A_s, $s \in \mathbb{N}$, such that for all s, $A_s \subseteq A_{s+1}$, and $A = \bigcup_s A_s$.*

Notice that property (iv) is almost effective countability, as in §3.4, but not quite.

Proof. The proofs that (ii), (iii), and (iv) imply (i) are essentially all the same. Dovetail all the $\varphi_e(x)$ computations, and whenever you see one converge, enumerate the preimage or the image involved depending on which case you're in. This is a computable procedure so the set produced will be computably enumerable.

(i)⇒(ii): Given A c.e., we define the following.

$$\psi(n) = \begin{cases} 1 & n \in A \\ \uparrow & n \notin A \end{cases}$$

We must show this is partial computable. To compute $\psi(n)$, begin enumerating A, a computable procedure. If n ever shows up, at that point output 1. Otherwise the computation never converges.

(i)⇒(iii): Again, given A c.e., note that we can think of its elements as having an order assigned to them by the enumeration: the first to be enumerated, the second to be enumerated, etc. (This will in general be different from their order by size.) Define the function using that:

$$\varphi(n) = (n + 1)^{st} \text{ element to be enumerated in } A.$$

(We use $n + 1$ to give 0 an image; this is not important here but we shall use it in the next part of the proof.) If A is finite, the

enumeration will cease adding new elements and φ will be undefined from some point on.

(i)⇒(iv): Suppose we have a nonempty c.e. set A. If A is infinite, the function φ from the previous paragraph is total, and A is its range. If A is finite, it is actually computable, so we may define

$$\hat{\varphi}(n) = \begin{cases} \varphi(n) & n < |A| \\ \varphi(0) & n \geq |A|. \end{cases}$$

□

Exercise 5.2.5. Prove the equivalence of part (v) with the rest of Proposition 5.2.4. Advice: prove (i) ⇔ (v) directly, and remember that in defining f you simply need to give a procedure that takes an arbitrary pair x, s and computes an answer in finite time.

Exercise 5.2.6. Prove the equivalence of part (vi) with the rest of Proposition 5.2.4. Again, prove (i) ⇔ (vi) directly, but this is simpler than Exercise 5.2.5. The sequence A_s is computable if there is a total computable function $f(s)$ giving the code of A_s for every s.

Exercise 5.2.7. Prove that every infinite c.e. set is the range of a *one to one* total computable function. This closes the gap in Proposition 5.2.4 (iv) with effective countability. Note that an enumeration of a set is allowed to list elements multiple times.

Every computable set is computably enumerable, but the reverse is not true. For example, we've seen that the halting set

$$K = \{e : \varphi_e(e)\downarrow\}$$

is c.e. (it is the domain of the diagonal function $\delta(e) = \varphi_e(e)$) but is not computable. What's the difference? Intuitively speaking, it's the waiting. If A is being enumerated and we have not yet seen 5, we do not know if that is because 5 is not an element of A or because it's going to be enumerated later. If we knew how long we had to wait before a number would be enumerated, and if it hadn't by then it never would be, then A would actually be computable: To find $\chi_A(n)$, enumerate A until you have waited the prescribed time. If n hasn't shown up in the enumeration by then, it's not in A, so output 0. If it has shown up, output 1. In the context of effective

countability, to determine whether n is in the set, you can compute $f(0)$, $f(1)$, $f(2)$, etc., for f the decoding function, but you will only get an answer if some $f(k)$ actually is equal to n.

Conversely, we can show a c.e. set is computable if we can cap the wait time.

Exercise 5.2.8. Prove that an infinite set is computable if and only if it can be computably enumerated in increasing order (that is, it is the range of a *monotone* total computable function).

Exercise 5.2.9. Prove that if A is c.e., A is computable if and only if $\overline{A}$ is c.e.

As in §5.1, we use subscripts to denote the state of affairs at a finite stage. In this case the stage-s version of A is the finite set A_s from Proposition 5.2.4 (vi). In fact, formally, we computably build finite sets $A_0 \subseteq A_1 \subseteq A_2 \subseteq \ldots$, and then define A to be $\bigcup_s A_s$.

It is straightforward to see that there are infinitely many sets that are not even c.e., much less computable. It is traditional to denote the domain of φ_e by W_e (and hence the stage-s approximation by $W_{e,s}$). The c.e. (including computable) sets are all listed out in the enumeration $W_0, W_1, W_2, \ldots$, which is a countable collection of sets. However, the power set of $\mathbb{N}$, which is the set of all sets of natural numbers, is uncountable. Therefore, in fact, there are not just infinitely many, but uncountably many sets that are not computably enumerable. These include all of the index set examples given in §4.5.

Exercise 5.2.10. Use Exercise 5.2.9 and the enumeration of c.e. sets, $\{W_e\}_{e\in\mathbb{N}}$, to give an alternate proof of the noncomputability of K.

Exercise 5.2.11. Prove that if A is computable and $B \subseteq A$ is c.e., then B is computable if and only if $A - B$ is c.e. Prove that if A is only c.e., $B \subseteq A$ c.e., we cannot conclude that B is computable even if $A - B$ is *computable*.

Exercise 5.2.12. Show that a function $f : \mathbb{N} \to \mathbb{N}$ is partial computable if and only if its graph $G = \{\langle x, y\rangle : f(x) = y\}$ is a computably enumerable set.

Exercise 5.2.13. Prove the *reduction property*: given any two c.e. sets A, B, there are c.e. sets $\hat{A} \subseteq A$, $\hat{B} \subseteq B$ such that $\hat{A} \cap \hat{B} = \emptyset$ and $\hat{A} \cup \hat{B} = A \cup B$.

Exercise 5.2.14. Prove that the set $\{\langle e, x, s\rangle : \varphi_{e,s}(x)\downarrow\}$ is computable.

Exercise 5.2.15. Prove that the set $\{\langle e, x\rangle : x \in W_e\}$ is c.e.; that is, prove that the c.e. sets are *uniformly enumerable*.

Exercise 5.2.16. Prove that the collection $\{(A_n, B_n)\}_{n\in\mathbb{N}}$ of all pairs of disjoint c.e. sets is uniformly enumerable.

Suggestion: enumerate triples $\langle n, i, x\rangle$, where n gives the pair of sets, $i \in \{0, 1\}$, and x is in A if $i = 0$ and B if $i = 1$. Alternatively, show that reduction, as in Exercise 5.2.13, may be done uniformly.

"All pairs of disjoint c.e. sets" means all possible pairings of c.e. sets such that the sets are disjoint. Note that as with the c.e. sets individually, the enumeration will contain repeats.

Exercise 5.2.17. Show that any infinite c.e. set contains an infinite computable subset.

Exercise 5.2.18. Show that any infinite set contains a noncomputable subset.

Exercise 5.2.19. Prove that if A and B are both computable (respectively, c.e.), then the following sets are also computable (c.e.).

(i) $A \cup B$

(ii) $A \cap B$

(iii) $A \oplus B := \{2n : n \in A\} \cup \{2n+1 : n \in B\}$, the *disjoint union* or *join*.

Exercise 5.2.20. Show that if $A \oplus B$, as defined above, is computable (respectively, c.e.), then A and B are both computable (c.e.).

Exercise 5.2.21. Two c.e. sets A, B are *computably separable* if there is a computable set C that contains A and is disjoint from B. They are *computably inseparable* otherwise.

(i) Let $A = \{x : \varphi_x(x)\downarrow = 0\}$ and $B = \{x : \varphi_x(x)\downarrow = 1\}$. Show that A and B are computably inseparable.

(ii) Let $\{(A_n, B_n)\}_{n\in\mathbb{N}}$ be the enumeration of all disjoint pairs of c.e. sets as in Exercise 5.2.16. Let $x \in A$ iff $x \in A_x$ and $x \in B$ iff $x \in B_x$, and show that A and B are computably inseparable. Hint: What if C were one of the B_n?

Exercise 5.2.22. (i) Show that if A is computably enumerable, the union $B = \bigcup_{e\in A} W_e$ is computably enumerable.

(ii) If A is computable, is B computable?

(iii) Can you make any claims about $C = \bigcap_{e\in A} W_e$ given the computability or enumerability of A?

Exercise 5.2.23. (i) Let X be a computable set, and define

$$A = \{n \in \mathbb{N} : (\exists m \in \mathbb{N})(\langle n, m\rangle \in X)\}.$$

$$B = \{n \in \mathbb{N} : (\forall m \in \mathbb{N})(\langle n, m\rangle \in X)\}.$$

Show that A is c.e. and B is co-c.e. (i.e., $\overline{B}$ is c.e.).

(ii) Let X and A be as above and assume A is noncomputable. Prove that for every total computable function f, there is some $n \in A$ such that $(\forall m)(\langle n, m\rangle \in X \;\Rightarrow\; m > f(n))$.

Exercise 5.2.24. Recall from §4.5 that the index set of all total functions is

$$\mathrm{Tot} = \{e : W_e = \mathbb{N}\} = \{e : \varphi_e \text{ is total}\}.$$

Prove that Tot is not computably enumerable.

Hint: Reread Theorem 5.2.4 (iv) and §4.2.

5.3. Aside: Enumeration and Incompleteness

Gödel's incompleteness theorem [30] is often summarized as "there are true but unprovable statements." This is an imprecise formulation, seemingly in conflict with his completeness theorem, which could be summarized as "a statement is true if and only if it is provable." The meaning of "true" in the two statements is different, and in this section, going light on details, we will elucidate the difference and discuss these theorems' relationship to the existence of noncomputable sets. For more details of incompleteness, I recommend Murawski [65];

9.2. Randomness

With a fair coin, any one sequence of heads and tails is just as likely to be obtained as any other sequence of the same length. However, our intuition is that a sequence of all heads or all tails, presented as the outcome of an unseen sequence of coin flips, smells fishy. It's just too special, too *nonrandom*. Here we'll present one way to quantify that intuitive idea of randomness and briefly explore some of the consequences and applications.

We will discuss randomness for infinite binary sequences. There are three main approaches to randomness.

(I) Compression: is there a short description of the sequence?

(II) Betting: can you get unboundedly rich by betting on the bits of the sequence?

(III) Statistics: does the sequence have any "special" properties?

Intuitively, the answer to each of those should be "no" if the sequence is to be considered random: a random sequence should be incompressible, unpredictable, and typical. There are different ways to turn these approaches into actual mathematical tests. The most fundamental are the following, in the same order as above.

(I) Kolmogorov complexity: How long an input does a prefix-free Turing machine need in order to produce the first n bits of the sequence as output? If it's always approximately n or more, the sequence is random. (Prefix-free TMs will be defined shortly.)

(II) Martingales: Take a function that represents the capital you hold after betting double-or-nothing on successive bits of a sequence (so the inputs are finite strings and the outputs are non-negative real numbers). Take only such functions that are computably approximable from below. Those are the c.e. martingales; if every such function has bounded output on inputs that are initial segments of your sequence, then the sequence is random.

(III) Martin-Löf tests: A computable sequence of c.e. sets $\{U_n\}$ such that the measure of U_n is bounded by 2^{-n} will have intersection of measure zero; this measure-zero set represents statistical "specialness." If your sequence is outside every such measure zero set, then it is random. (Measure will also be defined shortly.)

The nice thing about the implementations above is that they coincide [76]: A sequence is random according to Kolmogorov complexity if and only if it is random according to c.e. martingales if and only if it is random according to Martin-Löf tests. We call such a sequence *1-random*.

The changes made to these approaches to implement randomness in a different way tend to be restricting or expanding the collection of Turing machines, betting functions, or tests. We might take away the requirement that the machine be prefix-free, or we might allow it a particular oracle. In the opposite direction, we could require our machines not only to be prefix-free, but to obey some other restriction as well. We could allow our martingales to be more complicated than "computably approximable" or we could require they actually be computable. Finally, we could require our test sets have measure equal to 2^{-n} or simply require their measure limit to zero with no restriction on the individual sets, and we could play with the complexity of the individual sets and the sequence. The question of the "correct" cutoff, the one that defines "true randomness," is partly a philosophical one; we want a level high enough to avoid nonrandom behavior (if the cutoff is set too low, we could have a sequence that is called random but always has at least as many 1s as 0s in any initial segment, for instance), but otherwise as low as possible, since it does not seem reasonable to claim a string is compressible when, say, oracle $\emptyset''$ is required for the compression.

What does one do with this concept?

- Prove random sequences exist.
- Look at computability-theoretic properties of random sequences, considering them as sets.
- Compare different definitions of randomness.

- Consider *relative randomness*: if I know this sequence, does it help me bet on/compress/zero in on this other sequence?
- Look for sequences to which every random sequence is relatively random. Prove noncomputable examples exist.
- Extend the definition to other realms, such as sets of sequences.

We will explore only the Kolmogorov complexity approach. Good references for randomness are the books by Downey and Hirschfeldt [22], Nies [67], and Li and Vitányi [56]. For historical reading I suggest Ambos-Spies and Kučera [4], section 1.9 of Li and Vitányi [56], and Volchan [87].

9.2.1. Notation and Basics. For readability, some earlier definitions are reprinted here, along with new ones.

$2^{\mathbb{N}}$ is the collection of all infinite binary sequences and $2^{<\mathbb{N}}$ the collection of all finite binary strings. If you read computer science papers, you may see $\{0,1\}^*$ for $2^{<\mathbb{N}}$. The empty string is denoted λ, Λ, or $\langle\rangle$. 1^n is the string of n 1s and likewise for 0, and if σ and τ are strings, $\sigma\tau$ and $\sigma^\frown\tau$ both mean their concatenation. The notation $\sigma \subseteq \tau$ means σ is a (possibly non-proper) initial segment of τ or, in other words, that there is some string ρ (possibly equal to λ) such that $\sigma\rho = \tau$. Restriction of a string or sequence X to its length-n initial segment is denoted $X \upharpoonright n$.

In this field we tend to use n and the binary expansion of n interchangeably, so we would say the length of n, denoted $|n|$, is $\log n$ (all of our logarithms have base 2). If we are working with a string σ, then $|\sigma|$ is simply the number of bits in σ.

Infinite binary sequences, which we may clearly associate with sets, are also often referred to as *reals*, because we may view them as binary expansions of numbers between 0 and 1. This is not a bijection; any number with a terminating decimal expansion will pair with two sequences, one ending 1000... and one ending 0111.... Some intuition about numbers carries over to randomness; rational numbers will correspond to computable and hence nonrandom sequences. However, $\pi - 3$ will also give a computable binary sequence, because we can compute its decimal expansion to any number of significant

digits. It is in the cloud of anonymous transcendental numbers that the randoms live.

The real number intuition also helps with the topology. An *interval* is $[\sigma] = \{X : \sigma \subset X\}$, for any finite binary string σ. On the real line, it is $[0.\sigma 00\ldots, 0.\sigma 11\ldots]$. The *open sets* are countable unions of intervals; any interval (and hence finite union of intervals) is also closed, as the bracket notation implies. *Measure* is a way to assign a numerical value to a set to line up with some intuitive notion of size. We use the *coin-toss probability measure*; the measure of an interval $[\sigma]$ is $\mu([\sigma]) = 2^{-|\sigma|}$. It is the probability of landing inside $[\sigma]$ if you produce an infinite binary string by a sequence of coin flips from a fair coin. Intervals defined by longer strings have smaller measure; the sum of the measure of the intervals generated by all strings of a fixed length is 1, the measure of the whole space. The measure of the union of a pairwise-disjoint collection of intervals is the sum of the measure of the intervals.

9.2.2. Kolmogorov Complexity. Prefix-free Turing machines were suggested by Levin [55, 91] and later Chaitin [11] as the best way to approach the compressibility of strings. We will discuss the rationale after the definition and some results.

Definition 9.2.1. A Turing machine M is *prefix-free* if, for every pair of distinct strings $\sigma, \tau \in 2^{<\mathbb{N}}$ such that $\sigma \subset \tau$, M halts on at most one of σ, τ. Such a machine is generally taken to be *self-delimiting*, meaning the read/write head has only one-way movement; this does not restrict the class of functions computed by the machines.

What that means is that no string in the domain of M is a proper initial segment (or *prefix*) of any other string. Halting is therefore not contingent on knowing whether you've reached the end of the string: if you don't halt with the first n bits of input, either there is more input to be had or you will never halt.

Fortunately, there is a universal prefix-free machine. It can be taken to receive $1^e 0\sigma$ and interpret that as "run the e^{th} prefix-free

machine on input σ."[2] Call such a machine U. It is prefix-free because, in order to have $\sigma \subset \tau$, we must have $\sigma = 1^e 0 \sigma'$ and $\tau = 1^e 0 \tau'$ with $\sigma' \subset \tau'$, and since machine e is prefix-free, this cannot happen. We make the following definition.

Definition 9.2.2. The *prefix-free Kolmogorov complexity* of a string σ is

$$K(\sigma) = \min\{|\tau| : U(\tau) = \sigma\}.$$

Certainly if we hard-code a string into an input we can output any amount of it with just the constant cost of the program that says "print out the string that's listed here." We may have to do some additional work to put it into a prefix-free form, but this tells us the complexity of a string will have an upper bound related to the string's length. We say a string is random if we can't get much below that upper bound.

Definition 9.2.3. (i) A finite binary string σ is *random* if $K(\sigma) \geq |\sigma|$.

(ii) An infinite binary sequence X is 1-*random* if all of its initial segments are random, up to a constant. That is, $(\exists c)(\forall n)\ K(X \upharpoonright n) \geq n - c$.

Randomness for finite strings is problematic, because it depends on the enumeration of prefix-free Turing machines used in the definition of U. Randomness for infinite sequences is well-defined, however, because all differences in enumeration are swallowed up by the constant term.

9.2.3. The Size of K and Kraft's Inequality. To decide what it means to be incompressible, we needed to know something about the size of K. What upper bound can we assert about it, in terms of the

[2]Of course this assumes that the prefix-free machines can be enumerated. They can, by taking the enumeration of all Turing machines and modifying the machines that turn out to be non-prefix-free (compare Exercise 5.2.16). We work stagewise, with M being the given machine and P being the one we're building. At stage s, run M for s steps on the first s binary strings. If M is not prefix-free, then at some finite stage s^* M will halt on a string comparable to one on which M previously halted. Through stage $s^* - 1$ we let P exactly mimic M, and when (if) we see stage s^*, we define P to diverge on all remaining strings (including the one which witnessed that M was not prefix-free). If M is prefix-free, P will mimic it exactly.

length of σ? The following is part of a larger, more technical theorem, which I have trimmed in half. The proof uses a technique common in showing an upper bound on complexity: construct a specific machine that compresses the string by the desired amount.

Theorem 9.2.4 (Chaitin, [11]). *There is a c such that, for every σ of length n,*

$$K(\sigma) \leq n + K(n) + c.$$

Proof. Consider a prefix-free Turing machine T that computes $T(\tau\sigma) = \sigma$ for any τ and σ such that the universal prefix-free machine U gives $U(\tau) = |\sigma|$. Since T is prefix-free it has an index e in the enumeration of all prefix-free machines, and hence $U(1^{|e|}0e\tau\sigma) = \sigma$. That description has length $2|e| + |\tau| + |\sigma|$ or (if τ is as short as possible) $|\sigma| + K(|\sigma|) + 2|e|$, where e does not depend on σ, and its length certainly bounds the size of $K(\sigma)$. □

This upper bound leads to recursive further bounds:

$$K(\sigma) \leq n + |n| + ||n|| + |||n||| + \ldots.$$

Why do we not use this upper bound in our definition of randomness? Because there are no infinite string X and value c such that for all n, $K(X \upharpoonright n) \geq n + K(n) - c$. We could kludge by saying "for infinitely many n" instead of for all, but that's unsatisfying and difficult to work with. And, of course, the definition we gave for randomness is the one that lines up with Martin-Löf tests and martingales.

A (very rough) lower bound on K comes from the Kraft Inequality, a very useful tool in randomness. For a set to be prefix-free, there must be a lot of binary strings missing. Thus we would expect the length of these strings to grow rapidly, and they do.

Theorem 9.2.5 (Kraft Inequality, [49]). *Let $\ell_1, \ell_2, \ldots$ be a finite or infinite sequence of natural numbers. There is a prefix-free set of binary strings of length $\ell_1, \ell_2, \ldots$ if and only if*

$$\sum_n 2^{-\ell_n} \leq 1.$$

Proof. First, suppose we have a prefix-free set of finite binary strings σ_i with lengths ℓ_i. Consider

$$\mu\left(\bigcup_i [\sigma_i]\right).$$

Certainly this is bounded by 1, and since the set is prefix-free, the intervals are disjoint. Hence the measure of their union is the sum of their measures, and the inequality holds.

Now suppose we have a set of values $\ell_1, \ell_2, \ldots$ such that the inequality holds. We are not working effectively, so we may assume the set is nondecreasing. To find a prefix-free set of binary strings that have these values as their lengths, we carve up the complete binary tree, taking the leftmost string of length ℓ_i that is incomparable to the previously-chosen strings. [For example, if our sequence of values began 3, 4, 7, we would choose 000, 0010, 0011000.] Every binary string of length ℓ corresponds to an interval of size exactly $2^{-\ell}$, so by the inequality there will always be enough measure left to fit the necessary strings, and by our selection procedure all the remaining measure will be concentrated on the right and thus usable. □

This tells us that $K(\sigma)$ has to grow significantly faster than length (overall). The set of programs giving the strings σ is the prefix-free set here, and if those programs have length approximately $|\sigma|$, the sum $\sum_n 2^{-\ell_n}$ is essentially $2^{-1} + 2^{-1} + 2^{-2} + 2^{-2} + 2^{-2} + 2^{-2} + \ldots$, which diverges.

In the proof, we assumed the lengths we were given were in increasing order. In the effectivized version of Theorem 9.2.5 (Theorem 9.2.6, called the Kraft-Chaitin or KC Theorem), we can be given the required lengths of strings in any order and still create a prefix-free set with those lengths. In the proof of Theorem 9.2.5, if the string lengths are given out of order and misbehave enough, our procedure could take bites of varying sizes out of the tree so that when we get to length ℓ_n, although there is at least $2^{-\ell_n}$ measure unused in the tree, it is not all in one piece.

Theorem 9.2.6 (Chaitin [11,12], Levin [55])**.** *Let $\ell_1, \ell_2, \ldots$ be a collection of values (in no particular order, possibly with repeats, possibly*

finite) such that $\sum_i 2^{-\ell_i} \leq 1$. *Then from the sequence* ℓ_i *we can effectively compute a prefix-free set* A *with members* σ_i *of length* ℓ_i.

Proof. We present the proof in Downey and Hirschfeldt [22]. Assume that we have selected strings σ_i, $i \leq n$, such that $|\sigma_i| = \ell_i$. By induction, suppose also that we have a string $x[n] = 0.x_1x_2\ldots x_m = 1 - \sum_{j\leq n} 2^{-\ell_j}$, and that for every $k \leq m$ such that $x_k = 1$, there is a string $\tau_k \in 2^{<\mathbb{N}}$ of length k incomparable to all σ_j such that $j \leq n$ and all τ_j such that $j < k$ and $x_j = 1$.

Note that since $x[n]$ is the measure of the unchosen portion of $2^{<\mathbb{N}}$, the fact that there are strings of lengths corresponding to the positions of 1s in $x[n]$ means the remaining measure is concentrated into intervals of size at least as large as $2^{-\ell_{n+1}}$ for any ℓ_{n+1} which would allow satisfaction of the Kraft Inequality. Note also that the τ_k are unique and among them they cover the unchosen portion of $2^{<\mathbb{N}}$.

Now we select a string to correspond to ℓ_{n+1}. If $x_{\ell_{n+1}} = 1$, let $\sigma_{n+1} = \tau_{\ell_{n+1}}$ and let $x[n+1]$ be $x[n]$ but with $x_{\ell_{n+1}} = 0$; all τ_k for $k \neq \ell_{n+1}$ remain the same. If $x_{\ell_{n+1}} = 0$, find the largest $j < \ell_{n+1}$ such that $x_j = 1$. For the leftmost string τ of length ℓ_{n+1} extending τ_j, let $\sigma_{n+1} = \tau$. Let $x[n+1] = x[n] - 2^{\ell_{n+1}}$. As a result, in $x[n+1]$, $x_j = 0$, all of the x_k for $j < k \leq \ell_{n+1}$ are 1, and the remaining places of $x[n+1]$ are the same as in $x[n]$. Since τ was chosen to be leftmost in the cone τ_j, there will be strings of lengths $j+1, \ldots, \ell_{n+1}$ to be assigned as $\tau_{j+1}, \ldots, \tau_{\ell_{n+1}}$ (namely, $\tau_{j+i} = \tau_j 0^{i-1}1$), as required to continue the induction. □

One way to think of this is as a way to build prefix-free machines by enumerating a list of pairs of lengths and strings, with the intention that the string is described by an input of the specified length.

Theorem 9.2.7 (KC, restated)**.** *Suppose we are effectively given a set of pairs* $\langle n_k, \sigma_k\rangle_{k\in\mathbb{N}}$ *such that* $\sum_k 2^{-n_k} \leq 1$. *Then we can computably build a prefix-free machine* M *and a collection of strings (descriptions)* τ_k *such that* $|\tau_k| = n_k$ *and* $M(\tau_k) = \sigma_k$.

The KC Theorem allows us to implicitly build machines by enumerating "axioms" $\langle n_k, \sigma_k\rangle$ and arguing that the set $\{n_k\}_{k\in\mathbb{N}}$ satisfies the Kraft Inequality. On input τ, the machine enumerates axioms

while performing KC until such a time as τ is chosen to be an element of the prefix-free set, corresponding to some $\langle n_k, \sigma_k \rangle$. At that point (if it ever comes), the machine halts and outputs σ_k. This greatly expands the power of the proof technique used in Theorem 9.2.4.

9.2.4. Berry's Paradox, Formalized. *Berry's paradox* is often given as "the smallest number that cannot be described in fewer than thirteen words." That twelve-word phrase is (apparently) a description of the number, giving a self-contradictory statement. Philosophically, the resolution is in disallowing that as a description, since the claimed description references all descriptions (it is *impredicative*). If we formalize *description* in the sense of Kolmogorov complexity, however, we obtain meaningful results, including a proof of Gödel's incompleteness theorem. Durand and Zvonkin [23] give a very clear explanation of this material.[3]

First, let the function $t(n)$ be defined as $\max\{m \in \mathbb{N} : K(m) < n\}$. This exists because the number of possible descriptions of length $< n$ is finite. For all $x > t(n)$, $K(x) \geq n$; in particular, this holds of $t(n) + 1$. If t is a computable function, however, we need only n and some constant-size programming to get $t(n) + 1$, leading to a contradiction for sufficiently large n: $K(t(n) + 1) \leq K(n) + c_1 \leq \log n + K(\log n) + c_2$, where c_1 and c_2 do not depend on n. In fact, $t(n)$ grows faster than any computable function.

We may ramp this up to incompleteness by considering the provability of $K(x) \geq m$. We start with an axiomatizable sound theory T, which for our purposes here is simply a system for computably enumerating provable logical sentences (for details, see §§5.3 and 9.3). Suppose toward a contradiction that for such a T, $(\forall m)(\exists x)[(K(x) \geq m)$ is provable]. This gives an algorithm to find x from m, as follows: enumerate all theorems of T, and return x as soon as one of the form $K(x) \geq m$ is found. But then again, $K(x) \leq K(m) + c_1 \leq \log m + K(\log m) + c_2$ gives a contradiction for sufficiently large m.

This gives an entire collection of unprovable statements, namely $K(x) \geq m$ for any sufficiently large m. The function t and the provability discussion formalize Berry's Paradox in two ways. For t we

[3]Note that they use plain complexity (§9.2.6) but denote it by K.

use "the smallest integer n such that $K(n) > m$;" this does not give a description in the sense of Kolmogorov complexity, so there is no paradox to t existing (it simply can't be computable, or it *would* give a Kolmogorov description). However, we then changed it to "the integer n corresponding to the first theorem of the form $K(n) \geq m$ in the enumeration of theorems." That description did land us in the realm of Kolmogorov complexity and hence paradox, with the conclusion that for some m no such n exists.

Another result of formalizing Berry's Paradox in the manner of t above is the following. Recall that a *simple* set is a c.e. set A such that $\overline{A}$ is infinite but contains no infinite c.e. subsets.

Theorem 9.2.8 (Kolmogorov [46,47])**.** *The set of nonrandom numbers is simple.*

Proof. The set for which we need to prove simplicity is $A = \{x : K(x) < |x|\}$. If we dovetail the computations of a universal prefix-free Turing machine U, for any nonrandom x, we will eventually see U output x on an input of length $< |x|$. At that point we can put x into A, so A is c.e. We know the set of random numbers is infinite, so $\overline{A}$ is infinite.

We now show that every infinite c.e. set W_e contains a nonrandom element. Let i be such that $\varphi_i(\langle e, n\rangle)$ is the n^{th} element enumerated into W_e, $x_{e,n}$, if $|W_e| \geq n$, and undefined otherwise. Let $h(e, n)$ be the string that instructs U to emulate $\varphi_i(\langle e, n\rangle)$; h is a total computable function. Note that $h(e, n)$ is a description of $x_{e,n}$.

Set $t(n) = \max_{e \leq n} h(e, n)$; t is also total computable. For any index e, $t(n)$ will take $h(e, n)$ into account on all but finitely many values of n. We will use t to define a subset of W_e such that for some n, h gives a short description of the n^{th} element of the subset. Since that number is also in W_e itself, W_e contains a nonrandom element.

Given W_e, enumerate a subset Y so that the n^{th} element of Y, y_n, is greater than $t(n)$. If W_e is infinite, for any n it will contain elements larger than $t(n)$. When we see such an element enumerated we can put it into Y, which will therefore be infinite.

Since Y is c.e., it is $W_{\hat{e}}$ for some $\hat{e}$. However, by the choice of $y_n > t(n)$ and the fact that for almost all n, $t(n) \geq h(\hat{e}, n)$, we know

there is some n such that $x_{\hat{e},n} = y_n > t(n) \geq h(\hat{e}, n)$. For that n, $h(\hat{e}, n)$ gives a short description of $x_{\hat{e},n}$, so $x_{\hat{e},n}$ is a nonrandom element of $W_{\hat{e}} = Y$ and hence of W_e. □

This result gives incompleteness for many theories simultaneously.

Corollary 9.2.9. *There exists a computably enumerable set B with infinite complement such that for all axiomatizable sound theories T there are only finitely many n such that $n \notin B$ is true and provable in T.*

Proof. Let B be the set of nonrandom numbers. Since T is axiomatizable we can enumerate the set D of elements provably in the complement of B. However, since B is simple, D must be finite. □

Kolmogorov complexity may also be used to prove Gödel's second incompleteness theorem (see §9.5), but on that topic I will merely point you to the paper by Kritchman and Raz [50].

9.2.5. Halting Probability. Just as the halting problem is a fairly explicit noncomputable set, we may use halting to give a fairly explicit random number. This number is the *halting probability*, typically called (Chaitin's) Ω.

Before we define Ω, a brief discussion of degree. Every degree $\geq \mathbf{0'}$ contains a random set, and for every degree $\boldsymbol{d}$ there is a degree $\boldsymbol{d_1} \geq \boldsymbol{d}$ that contains a random. However, unless $\boldsymbol{d} \geq \mathbf{0'}$, there will also be some degree $\boldsymbol{d_2} \geq \boldsymbol{d}$ that does not contain a random.

No computably enumerable set is random. It is clear that any computable set cannot be random, so we need only consider noncomputable and hence infinite c.e. sets. From a betting perspective, this allows us to wait to place bets until we see a new 1 in the sequence ahead of where we have thus far placed bets. At that time we can bet evenly on 0 and 1 for the bits up to that new 1, neither gaining nor losing money, and then place all of our money on 1. Since we can thereby double our money infinitely many times, the sequence cannot be random.

However, it is possible for a set of c.e. degree to be random, as long as that degree is $\emptyset'$. It is even possible for the set itself to be left-c.e., as defined in §6.3.[4] Ω is such a random. For U a universal prefix-free Turing machine, Ω is defined as a real:

$$\Omega = \sum\{2^{-|\sigma|} : \sigma \in 2^{<\mathbb{N}} \ \& \ U(\sigma)\downarrow\}.$$

In §6.3, left-c.e. sequences were defined as those possessing an approximation that increases in value, and here that is built into the definition: as we see computations halt, we add the measure of the input string to the current value of Ω.

Since U must halt on some σ, $\Omega > 0$. By the Kraft Inequality 9.2.5, $\Omega \leq 1$. In fact, U's domain is a subset of $\{1^e0\tau : e \in \omega\}$, and for e the index of a Turing machine with empty domain $U(1^e0\tau)\uparrow$ for all τ. Therefore $\Omega < 1$.

Claim 9.2.10. *Let σ be a binary string of length at most n. From $\Omega \upharpoonright n$ we may effectively determine whether $U(\sigma)\downarrow$.*

Proof. Observe that $\Omega \upharpoonright n \leq \Omega < \Omega \upharpoonright n + 2^{-n}$. We dovetail the computations of U on all inputs and approximate Ω by Ω_s, which begins as zero.

If $U(\tau)\downarrow$ at stage s, let $\Omega_s = \Omega_{s-1} + 2^{-|\tau|}$. Eventually we will see $\Omega_s \geq \Omega \upharpoonright n$. If σ is not among the programs for which we have already seen halting, it will never halt, as it would add at least 2^{-n} to Ω_s, making it larger than Ω. □

Chaitin's Ω has some philosophical interest as "the number of Wisdom" (Bennett and Gardner [9]): if you know $\Omega_{1:10000}$ and have an axiomatizable mathematical theory expressible in 10,000 or fewer bits, you can find whether its statements are true, false, or independent. This includes Goldbach's Conjecture, the Riemann Hypothesis, and most other conjectures in mathematics which would be refutable with finite counterexamples – the programs looking for such counterexamples will or will not halt, and Ω knows their behavior.

However, the function $t(n)$, giving the amount of time needed to find all halting programs of length less than n from $\Omega \upharpoonright n$, grows

[4] If you read the literature, be aware that in randomness a left-c.e. real is sometimes referred to as just a c.e. real.

faster than all computable functions, so knowing Ω gives no practical help.

Before leaving Ω we should prove it is random.

Claim 9.2.11. *Ω is K-random; that is,* $(\exists c)(\forall n)\ K(\Omega \upharpoonright n) \geq n - c$.

Proof. We demonstrate that there is a computable function φ such that $K(\varphi(\Omega \upharpoonright n)) > n$. The difference in K-complexity between σ and $f(\sigma)$ for any computable f is only a constant, so this will prove the claim.

From Claim 9.2.10 it follows that from $\Omega \upharpoonright n$, one may calculate all programs σ of length at most n on which U halts. There will be, therefore, some τ_n that is not yet computed by any such σ, and therefore such that $K(\tau_n) > n$. Let $\varphi(\Omega \upharpoonright n) = \tau_n$. □

9.2.6. Why Prefix-Free? We could define the complexity of σ as the minimum length input that produces σ when given to the standard universal Turing machine, rather than the universal prefix-free Turing machine. That is the *plain Kolmogorov complexity* of σ, denoted $C(\sigma)$.[5] However, as a standard for complexity it has some problems, even at the level of finite strings.

The first undesirable property of C is non-subadditivity: for any c there are x and y such that $C(\langle x, y\rangle) > C(x) + C(y) + c$. K, on the other hand, *is* subadditive, because with K we can concatenate descriptions of x and y, and the machine will be able to tell them apart: the machine can read until it halts, assume that is the end of x's description, and then read again until it halts to obtain y's description. Some constant-size code to specify that action and how to encode the x and y that result, and we have $\langle x, y\rangle$.

The second undesirable property is nonmonotonicity on prefixes: the complexity of a substring may be greater than the complexity of the whole string. For example, a power of 2 has very low complexity, so that if $n = 2^k$ then $C(1^n) \leq \log\log n + c$ (i.e., a description of k, which is no more than $\log k$ in size, plus some machinery to take

[5]Historically there has been some variation in randomness notation. The older the paper, the more likely you are to see K for plain complexity, and either H or KP for prefix-free complexity. For a time, prefix-free complexity was also unfortunately referred to as "prefix complexity."

powers and print 1s). However, once k is big enough, there will be numbers smaller than n that have much higher complexity because they have no nice concise description in terms of powers of smaller numbers or similar. For such a number m, $C(1^m)$ would be higher than $C(1^n)$ even though 1^m is a proper initial segment of 1^n.

The underlying problem is that $C(\sigma)$ contains information about the length of σ (that is, n) as well as the pattern of bits. For most n, about $\log n$ of the bits of the shortest description of σ will be used to determine n. What that means is that for simple strings of the same length n, any distinction between the pattern complexity of the two strings will be lost to the domination of the complexity of n.

Another way of looking at it is that C allows you to compress a binary sequence using a ternary alphabet: 0, 1, and "end of string." That's not a fair measure of compressibility, and as stated above, it leads to some technical as well as philosophical problems. The main practical argument for K over C, though, is that K gives the definition that lines up with the characterizations of randomness in terms of Martin-Löf tests and martingales.

9.2.7. Relative Randomness and K-Triviality. Our final topic is a way to compare sets to each other, more finely grained than saying both, one, or neither is random. For example, the bit-flip of a random sequence is random, but if we are given the original sequence as an oracle, its bit-flip can be produced by a constant-size program. Therefore no sequence's bit-flip is random *relative to* the original sequence.

Definition 9.2.12. (i) The *prefix-free Kolmogorov complexity of σ relative to A* is $K^A(\sigma) = \min\{|\tau| : U^A(\tau) = \sigma\}$.

(ii) A set or sequence B is *A-random* (or 1-A-random) if

$$(\exists c)(\forall n)[K^A(B \upharpoonright n) \geq n - c].$$

It should be clear that if B is nonrandom, it is also non-A-random for every A. Adding an oracle can never *increase* the randomness of another string; it can only derandomize. That is, if RAND is the set of all 1-random reals and RAND^A is the set of all A-random reals, then for any A, $\text{RAND}^A \subseteq \text{RAND}$. The question is then for which A

equality holds; certainly for any computable A it does, but are there others? Hence we have the following definition, a priori perhaps a duplication of "computable."

Definition 9.2.13. A set A is *low for random* if $\text{RAND}^A = \text{RAND}$.

A low for random set clearly cannot itself be random, because any sequence derandomizes itself and its infinite subsequences. The term "low" is by analogy with ordinary computability theory, where A is low if the halting problem relativized to A is unchanged in degree from the nonrelativized halting problem. As there exist noncomputable low sets, Kučera and Terwijn [52] have shown that there exist noncomputable low for random sets.

A low for random sequence is one that K cannot distinguish from a computable sequence, with respect to its usefulness as an oracle. Another way for K to distinguish between sequences is their initial segment complexity. Hence we have the following definition.

Definition 9.2.14. A real X is *K-trivial* if the prefix-free complexity of its length-n initial segments is bounded by the complexity of n; that is, $(\exists c)(\forall n)(K(X \upharpoonright n) \leq K(n) + c)$.

The question is, again, whether there are any noncomputable K-trivial reals. Certainly all computable reals X are such that $K(X \upharpoonright n) \leq K(n) + c$; the constant term holds the function that generates the initial segments of X, and then getting an initial segment is as simple as specifying the length you want.

Theorem 9.2.15 (Zambella [90], after Solovay [83])**.** *There is a noncomputable c.e. set A such that $(\exists c)(\forall n)(K(A \upharpoonright n) \leq K(n) + c)$.*

The truly remarkable thing is that these are the same class of reals: a real is low for random if and only if it is K-trivial. The proof is extremely difficult, involving work by Gács [29], Hirschfeldt, Nies, and Stephan [38], Kučera [51], and Nies [66].

9.3. Some Model Theory

Both computable model theory (§9.4) and reverse mathematics (§9.5) involve model theory, another area of mathematical logic. Classes on