6 Decidability

In Chapter 5 we saw how to formally define an algorithm as a program in the RAM model, and gave some intuition for why this definition allows us to describe any algorithm that we can write on an actual computer.

In this chapter we will explore the topic of computability. In Section 6.1 we discuss how we can encode objects as strings so that we can create algorithms with inputs or outputs of whatever type we want, including algorithms which take other algorithms as input. In Section 6.2 we will define decidability and look at how to prove that a language is decidable. Finally, in Section 6.3 we will see our first example of a problem that algorithms cannot solve.

6.1 String Encodings and Universal Machines

A RAM model program takes as input a single string. This may seem limiting, since many functions we care about involve inputs other than strings (such as integers, arrays, or graphs). It turns out, however, that we don't lose much by restricting ourselves to string inputs, because all kinds of other objects can be *encoded* as strings. For some object A, we use $\langle A \rangle$ to denote the string representation of A. Similarly for a sequence of objects $A_1, \ldots, A_k$ we use $\langle A_1, \ldots, A_k \rangle$ to denote a single string encoding all of the objects.

Note that there are a few objects we care about which cannot be encoded as strings. In particular, we cannot encode arbitrary real numbers as strings. Intuitively, this is because real numbers can have infinitely long decimal representations with no pattern to the digits, and our algorithms can only take *finite* input strings. Thus, our algorithms cannot take arbitrary real numbers as inputs or outputs, only ones which can be expressed as finite strings (e.g. rational numbers, which can be encoded using their numerator and denominator, even if their decimal representations would be infinitely long).

There are many different ways to encode the same object as a string (for example, we could represent integers in binary or in base-10 or in some more exotic system), and in future chapters it will sometimes be important to specify which one we're using. For this chapter, however, we are only focused on the correctness of our algorithms/programs, not their runtime or space usage, so the particular encoding scheme that we use will generally not matter. Therefore we will not add unnecessary clutter by formally specifying what encoding schemes we are using.

Whenever we write that an algorithm takes a particular kind of object as input, we implicitly mean that the algorithm interprets its input as an encoding of that object (i.e. if we say that an algorithm takes an integer as input, the actual input is the string $\langle n \rangle$, but the first thing that our algorithm does is convert this string into an integer).

When we say that an algorithm returns a particular object, we mean that the algorithm outputs the encoding of that object. For example, when working with the alphabet $\Sigma = \{0, 1\}$, the instruction "Return True" may actually be implemented by outputting the string "1", but we will not get into these encoding details in our pseudocode.

One important class of objects which we can encode with strings is algorithms themselves. After all, the source code of a program in the RAM model (or any programming language) is just a long string. This means that we can write algorithms that take the encodings of other algorithms as input (you probably use such a program frequently—one example is a compiler). Just as for other objects, we will use $\langle P \rangle$ to represent the encoding of the algorithm P as a string (we will adopt the convention of naming generic algorithms with P for "program" to avoid confusion with arrays which we often call A).

A **universal machine** is an algorithm that takes as input the encoding of an algorithm and a string, and simulates running the algorithm on the string (You can think of a universal machine as the RAM model's equivalent of the Python interpreter). More formally, a universal machine takes as input a string in the form $\langle P, x \rangle$, where P is an algorithm and x is a string. If $P(x)$ outputs a string, then the universal machine outputs the same string. If $P(x)$ loops forever (never reaching an "Output" instruction) then the universal machine also loops forever when run on $\langle P, x \rangle$.

It's not obvious that a universal machine should exist. Describing how to construct a universal machine was one of the major contributions of Alan Turing's paper "On Computable Numbers, with an Application to the Entscheidungsproblem", which is also where he first defined Turing machines. At a very high level, our universal machine can use an integer variable to keep track of which line of P it is currently simulating, and designate one section of its memory for storing the code of P, another section to store the memory of P, and a final section for scratch space used to help with tasks like converting the memory addresses used by P into addresses in the universal machine's memory. Formally describing a universal machine is tedious and not the focus of this course, however, so we omit the details. For the purposes of algorithms you write in this course, you may assume that a universal machine exists and therefore that algorithms which you write can simulate the execution of an algorithm given its encoding.

It's important to note that while simulating P with a universal machine preserves the output, it does not necessarily preserve the runtime. Running a universal machine on $\langle P, x\rangle$ will generally be asymptotically slower than directly running $P(x)$. Nevertheless, since we are not concerned with runtime in this chapter, we will allow ourselves to treat an algorithm which we have read from its source code like a regular function we have written and include in our pseudocode lines like "result $\leftarrow P(x)$" as an abbreviation for "Simulate running $P(x)$ and store the output in result".

6.2 Decidability

For the rest of this chapter, we will focus on algorithms that return only True or False.

We define a **language** as a set of strings. That is, for a fixed alphabet Σ, a language is a subset of Σ^*. For example, if our alphabet is $\{0, 1\}$, then $\{010, 1111, \varepsilon\}$ is a language of size 3, and $\{1^n 0 \mid n \in \mathbb{Z}^+\}$ is a language of infinite size. For an algorithm P, we define the **language of** P, denoted $L(P)$ by

$$L(P) := \{x \in \Sigma^* \mid P(x) \text{ returns True}\}.$$

If L is a language and P is an algorithm with $L(P) = L$, we say that P **accepts** L. Note that P may return False or loop on strings not in L. We say that a language L is **acceptable** or **semi-decidable** if there exists an algorithm that accepts L.

If L is a language, P is an algorithm that accepts L, and P returns False on all strings in $\Sigma^* \setminus L$, we say that P **decides** L. We say that a language L is **decidable** if there exists some algorithm that decides L. Note that every decidable language is decided by infinitely many different algorithms (we can always add extra pointless lines of code, e.g. increment a variable then decrement it). An algorithm which always returns True or False (never gets stuck in an infinite loop) is called a **decider**. Note that if P is a decider then P decides $L(P)$, while if P is not a decider then P merely accepts $L(P)$.

The simplest way to prove that a language is decidable is to give an algorithm and prove that the algorithm decides the language.

Theorem 29. *Let*

$$L := \{\langle n, p\rangle \mid n, p \in \mathbb{Z}^+, \text{ and there are at least } p \text{ prime numbers between 1 and } n\}$$

Show that L is decidable.

Proof. We show that L is decidable by giving an algorithm to decide it.

```
L-Decider(n, p)
    count ← 0
    For i from 2 to n:
        prime ← True
        For j from 2 to i − 1:
            If i mod j = 0:
                prime ← False
        If prime:
            count ← count+1
    If count ≥ p:
        Return True
    Return False
```

We note that, by definition, a positive integer $i \geq 2$ is prime if and only if it is not divisible by any positive integer between 2 and $i-1$, or equivalently if $i \mod j \neq 0$ for every j between 2 and $i-1$. Therefore, after the outer for loop of L-Decider the variable count stores the number of primes between 2 and n. This is the same as the number of primes between 1 and n, since 1 is not prime. Since L-Decider returns True if and only if count $\geq p$, L-Decider returns True if $\langle n, p \rangle \in L$ and returns False otherwise. Thus, L-Decider decides L. □

There are several things to note about this proof:

- The definition of a decidable language is one with an algorithm which decides it. Therefore to prove that a language is decidable, we can give an algorithm and then prove that this algorithm correctly decides the language.
- It's not sufficient to simply give the algorithm. We need to be able to prove that our algorithm actually does what we claim it does.
- Although we formally define an algorithm as a RAM model program, to prove that an algorithm exists it is sufficient to give high level pseudocode, since as we discussed in Section 5.6, any feature we can implement in a regular programming language we can also implement in the RAM model.
- Recall that an algorithm always takes a string as input. When we describe an algorithm as taking some other kind of input, we implicitly assume that the algorithm returns False when given a string that does not encode an object of this form. For example, in our proof we assumed that L-Decider takes as input a pair of positive integers, n and p (because this is what elements of L encode). What we mean by this is that L-Decider will begin by checking if its input encodes a pair of positive integers and return False if it does not.
- Any algorithm which decides a language is sufficient to prove that the language is decidable. It doesn't have to be a fast or clever algorithm (we also don't have to actually calculate the runtime of the algorithm). Often, the easiest way to prove a problem is decidable is some kind of "brute force" algorithm. There are much faster algorithms for this primality problem, but using one of them would just be making our correctness proof harder.

Theorem 30. *The language*

$$\{\langle P, x, k \rangle \mid P \text{ is an algorithm}, x \in \Sigma^*, k \in \mathbb{Z}^+, \text{ and } P(x) \text{ halts after at most } k \text{ steps}\}$$

is decidable.

Proof. Call this language L. We show that L is decidable by giving an algorithm to decide it.

```
L-Decider(P, x, k)
    Simulate P(x) for k steps
    If P(x) halted within the simulation:
        Return True
    Return False
```

We note that simulating $P(x)$ for only k steps can be done in finite time, even if $P(x)$ loops forever. To prove that L-DECIDER correctly decides L we need to show that $\forall\langle P, x, k\rangle \in L$, L-DECIDER$(P, x, k)$ returns True and $\forall\langle P, x, k\rangle \in \Sigma^* \setminus L$, L-DECIDER$(P, x, k)$ returns False. In this case both directions are straightforward. Let P be an algorithm, x be a string, and k be a positive integer.

- Suppose $\langle P, x, k\rangle \in L$. Then, by the definition of L, $P(x)$ halts after at most k steps. Therefore, when L-DECIDER simulates $P(x)$ for k steps, $P(x)$ will halt within the simulation, so L-DECIDER(P, x, k) will return True.
- Now suppose $\langle P, x, k\rangle \in \Sigma^* \setminus L$. Then by the definition of L, $P(x)$ is still running after k steps. Therefore, when L-DECIDER simulates $P(x)$ for k steps, $P(x)$ will not halt, and so L-DECIDER$(\langle P, x, k\rangle)$ will return False.

□

In this proof our algorithm included the line "Simulate $P(x)$ for k steps". From our discussion of universal machines, we know that simulating another algorithm (including simulating another algorithm for a fixed number of steps) is something that algorithms can do. Therefore, we will allow this "simulation" operation as a single line of pseudocode, even if actually implementing it in the RAM model would be complicated. Similarly since we are not discussing the details of encodings that we are using in this chapter, we will allow algorithms to convert between objects and their encodings without specifying how this is done. For example, if we have an algorithm that takes as input $\langle P\rangle$, the encoding of an algorithm P, we could write a line like

$$n \leftarrow \text{ the number of instructions in } P$$

without discussing how we obtained the number of instructions from the string $\langle P\rangle$.

It's important to understand that whether or not a language is decidable does not depend on whether we *know* an algorithm which decides the language, and in fact it is possible to prove that a language is decidable without constructing a specific algorithm which decides it, as the following problem demonstrates (we call such a proof "non-constructive").

Theorem 31. *Let*

$$L = \{\langle n\rangle \mid n \in \mathbb{Z}_{\geq 0} \text{ and the base-10 expansion of } \pi \text{ contains } n \text{ consecutive } 7s\}$$

Then L is decidable.

Proof. We consider two cases, and show that in either case, an algorithm that decides this language exists.

- First, suppose that for every positive integer k, there is a sequence of k consecutive 7s somewhere in the decimal expansion of π. In this case, L is decided by the following algorithm:

```
L-DECIDER(n)
  Return True
```

- Now suppose that there is some integer k such that the decimal expansion of π does not contain a sequence of k consecutive 7s. Choose the minimum such k. By our choice of k, π contains a sequence of $k-1$ consecutive 7s, and this sequence contains within it a sequence of i consecutive 7s for each $i < k$ (simply take the first i elements of the sequence). On the other hand, π cannot contain a sequence of i consecutive 7s for any $i \geq k$, because this sequence would contain a sequence of k consecutive 7s, and by our choice of k, no such sequence exists. Therefore, L is decided by the following algorithm:

```
L-DECIDER(n)
  If n < k :
      Return True
  Return False
```

Note that we do not know which of these cases holds (and in the second case we don't know the value of k), and thus we don't know which of these algorithms decides L. Nevertheless, we know that one of the algorithms we described above must decide L, so L is decidable even if we aren't sure which of the algorithms we presented is the one that actually decides it.

Sidenote: Even though π is transcendental, it's actually an open problem whether or not its decimal expansion contains strings of 7s (or any other digit) of arbitrary length. □

6.3 The Halting Problem

Before Turing's famous 1936 paper it was widely believed that every problem could be solved by a sufficiently clever algorithm—i.e. that every language was decidable. One of Turing's largest contributions was proving that there exist undecidable languages—problems that algorithms cannot solve. Formally we say that a language is **undecidable** if it is not decidable. In this section we will see the first and most famous undecidable language and see how to prove that it is undecidable.

We define

$$\text{HALTS} = \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ halts}\}.$$

That is, HALTS is a language of algorithm-string pairs where the algorithm halts (outputs a value) when run on the string. This would be a useful language to have a decider for. For instance, when writing a homework autograder, it would be useful to have a way to distinguish between programs that loop forever versus ones that are very slow.

Intuitively, we might believe that HALTS should be decidable by some kind of brute force approach. Perhaps we can just run the algorithm for a long time and see if it ever ends up on the same instruction with the same data in its variables and in memory? Repeating all of these things would indeed imply that the algorithm is looping. Unfortunately since a RAM model program can expand its memory an arbitrary number of times, it's possible for a RAM model program to loop without ever repeating the same exact memory contents (A trivial example is a program that just repeatedly calls Malloc forever). There is no way we can brute force over the infinitely many potential memory configurations that may occur while running a given algorithm, so this approach to deciding HALTS does not work.

Stronger evidence that the Halting problem must be hard comes from the fact that an algorithm which decides HALTS could solve essentially every open question in mathematics. Proofs in mathematics can be written in an extremely formal language which is verifiable by a computer. For example, a number of important mathematics proofs have been written and verified in Lean. In principle, any mathematical proof could be written in Lean (although the proof might be very long), and we could construct an algorithm that takes a string and determines whether or not it is a Lean proof of a particular theorem. For example, we could create the following algorithm:

```
RIEMANN(x)
  For i from 0 to ∞
      For each string s ∈ Σ^i
          If s encodes a proof of the Riemann Hypothesis in Lean:
              Return True
```

The Riemann Hypothesis is one of the most famous unsolved problems in mathematics, with a one million dollar prize for whoever solves it. If the Riemann hypothesis is provable (e.g. because it is true), then it must have a Lean proof, which has some finite length n. Since there are only finitely many strings of length at most n, our algorithm RIEMANN will check all of them in finite time, find the proof of the Riemann hypothesis and return True. On the other hand if the Riemann Hypothesis is false, then there cannot be a proof of it, so RIEMANN will loop forever. Thus, if we could decide

whether or not RIEMANN halts (the choice of input does not matter here, since RIEMANN ignores its input string x), then we could determine whether or not the Riemann hypothesis is true. There's nothing special about the Riemann hypothesis for this purpose either, we could construct a similar algorithm for any mathematical conjecture. The idea of a single program which can solve every mathematical conjecture may seem a bit too good to be true.

Indeed, Turing proved the following theorem:

Theorem 32. HALTS *is undecidable.*

Proof. We prove this by contradiction.

Suppose for contradiction that HALTS is decidable. Then, by the definition of a decidable language, there exists an algorithm HALTS-DECIDER which decides HALTS. The key concept we will use to obtain a contradiction is running an algorithm on its own encoding (source code). We construct a new algorithm CONTRARY which takes as input an algorithm P, then asks HALTS-DECIDER whether the algorithm halts when run on its own source code. If HALTS-DECIDER says that $P(\langle P \rangle)$ halts, then CONTRARY enters an infinite loop. If HALTS-DECIDER says that $P(\langle P \rangle)$ loops forever, then CONTRARY immediately halts.

```
CONTRARY(P)
  If HALTS-DECIDER(P, ⟨P⟩) returns True:
      Loop forever
  Return True
```

Now consider what happens if we run CONTRARY($\langle$CONTRARY$\rangle$). Then CONTRARY will call HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$). We consider two cases:

- Suppose that $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \in$ HALTS. Then, since HALTS-DECIDER is a decider for HALTS, HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$) returns True. Thus, CONTRARY($\langle$CONTRARY$\rangle$) will reach the "Loop forever" line and loop forever. Therefore $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \notin$ HALTS. This contradicts what we assumed at the start of this case, so this case cannot occur.
- Now suppose that $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \notin$ HALTS. Then, since HALTS-DECIDER is a decider for HALTS, HALTS-DECIDER(CONTRARY, $\langle$CONTRARY$\rangle$) returns False. Therefore, CONTRARY($\langle$CONTRARY$\rangle$) will reach the "Return True" line and halt.
  Thus, $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle \in$ HALTS. This contradicts what we assumed at the start of this case, so this case cannot occur either.

Thus, whether $\langle$CONTRARY, $\langle$CONTRARY$\rangle\rangle$ is in HALTS or not, we get a contradiction, meaning that our initial assumption that HALTS-DECIDER exists must have been false, and HALTS is undecidable. □

Note that HALTS is semi-decidable, as shown by the following simple algorithm that accepts HALTS:

```
HALTS-ACCEPTOR(P, x)
  Run P(x)
  Return True
```

The "Run $P(x)$" line will eventually finish running if and only if $P(x)$ halts. Thus, if $P(x)$ halts, HALTS-ACCEPTOR returns True, and if $P(x)$ loops forever then so does HALTS-ACCEPTOR.

6.4 Comprehension Questions

1. True or false? Every acceptable language is decidable.

2. Suppose that U is a universal machine. We define two languages:

$$L_1 := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns True}\}$$
$$L_2 := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ halts}\}$$

   Which of the following is true?

   (a) U decides the language L_1.

   (b) U decides the language L_2.

   (c) U accepts the language L_1, but does not decide it.

   (d) U accepts the language L_2, but does not decide it.

   (e) U does not accept or decide any of these languages.

3. True or false? The language $L := \{\langle n\rangle : n \in \mathbb{Z}_{\geq 0}$ and the base-10 expansion of π contains at most n consecutive 0s$\}$ is decidable.

4. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ (i.e. the digits 0-9 as well as a decimal point). Which of the following sets are languages? (Select all that apply)

   (a) Σ^*

   (b) $\mathbb{R}^+$

   (c) $\mathbb{Z}^+$

   (d) $\{\langle x\rangle \mid x \in \mathbb{Z}^+\}$

5. True or false? The language HALTS is acceptable.

6. Two of the following four languages are decidable. Determine which two.

   (a)

$$\text{HAS-MULT} := \{\langle P\rangle \mid P \text{ is an algorithm which includes an instruction of the form } i \leftarrow j * k\}$$

   (b)

$$\text{USES-MULT} := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ executes an instruction of the form } i \leftarrow j * k\}$$

   (c) NOT-HALTS $:= \Sigma^* \setminus$ HALTS

   (d) HALTS-FAST $:= \{\langle P, x\rangle \mid P$ is an algorithm, $x \in \Sigma^*$ and $P(x)$ halts after at most 100 steps$\}$

6.5 Exercises

1. Prove that if L_1 and L_2 are two decidable languages then $L_1 \cup L_2$ is decidable.

2. Prove that there exist two undecidable languages L_1 and L_2 such that $L_1 \cup L_2$ is decidable.

3. Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. Prove that if L and $\Sigma^* - L$ are both semi-decidable, then L is decidable.

4. Prove that the following language is decidable:

   NO-MAL-HALTS $:= \{\langle P, x\rangle \mid P$ is an algorithm, $x \in \Sigma^*$ and $P(x)$ halts without calling Malloc.$\}$

5. A graviton is a hypothetical elementary particle. Physicists are currently not sure whether or not gravitons exist. Prove that the following language is decidable (no knowledge of physics required):

$$\text{GRAVITONS} := \begin{cases} \{\text{“yes”}\} & \text{if gravitons exist} \\ \{\text{“no”}\} & \text{otherwise} \end{cases}$$

6. We say that a RAM model program P is “pretty small” if it satisfies the following conditions:

   - P's alphabet is $\Sigma = \{0, 1\}$
   - P has at most 411 instructions.
   - All integer constants used in P are between 0 and 411.

   Prove that the following language is decidable:

$$\text{SMALL-HALTS} := \{\langle P \rangle \mid P \text{ is a “pretty small” program and } P(\varepsilon) \text{ halts}\}$$

   (Hint: How many meaningfully distinct “pretty small” programs are there)