12 Computability

However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number of purely logical processes. . . This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason; for in mathematics there is no ignorabimus.
David Hilbert, 1900

In this chapter we consider the question of what problems can and cannot be solved by mechanical computation. This is the question of *computability*: a problem is *computable* if it can be solved by some algorithm; a problem that is *noncomputable* cannot be solved by any algorithm.

computability

Section 12.1 considers first the analogous question for declarative knowledge: are there true statements that cannot be proven by *any* proof? Section 12.2 introduces the *Halting Problem*, a problem that cannot be solved by any algorithm. Section 12.3 sketches Alan Turing's proof that the Halting Problem is noncomputable. Section 12.4 discusses how to show other problems are noncomputable.

12.1 Mechanizing Reasoning

Humans have been attempting to mechanize reasoning for thousands of years. Aristotle's *Organon* developed rules of inference known as *syllogisms* to codify logical deductions in approximately 350 BC.

syllogisms

Euclid went beyond Aristotle by developing a formal axiomatic system. An *axiomatic system* is a formal system consisting of a set of *axioms* and a set of *inference rules*. The goal of an axiomatic system is to codify knowledge in some domain.

axiomatic system

The axiomatic system Euclid developed in *The Elements* concerned constructions that could be drawn using just a straightedge and a compass.

Euclid started with five axioms (more commonly known as *postulates*); an example axiom is: *A straight line segment can be drawn joining any two points.* In addition to the postulates, Euclid states five *common notions*, which could be considered inference rules. An example of a common notion is: *The whole is greater than the part.*

Starting from the axioms and common notions, along with a set of definitions (e.g., defining a *circle*), Euclid proved 468 propositions mostly about geometry

proposition and number theory. A *proposition* is a statement that is stated precisely enough to be either true or false. Euclid's first proposition is: given any line, an equilateral triangle can be constructed whose edges are the length of that line.

proof A *proof* of a proposition in an axiomatic system is a sequence of steps that ends with the proposition. Each step must follow from the axioms using the inference rules. Most of Euclid's proofs are constructive: propositions state that a thing with a particular property exists, and proofs show steps for constructing something with the stated property. The steps start from the postulates and follow the inference rules to prove that the constructed thing resulting at the end satisfies the requirements of the proposition.

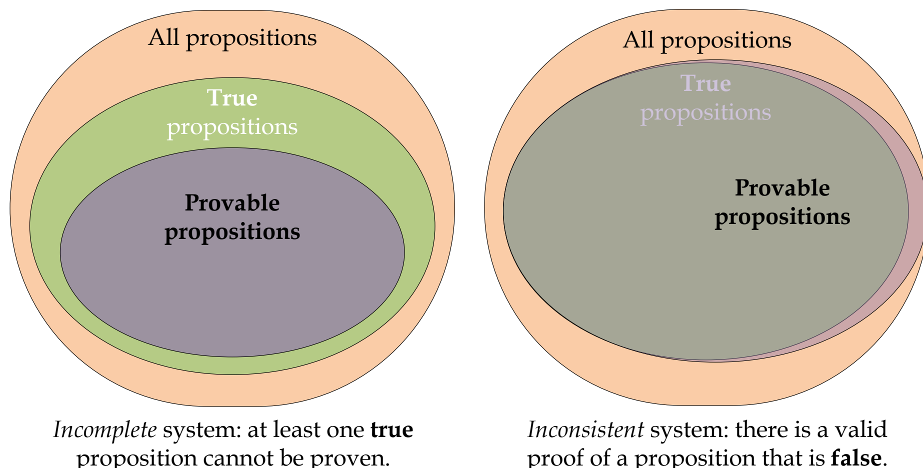
consistent A *consistent* axiomatic system is one that can never derive contradictory statements by starting from the axioms and following the inference rules. If a system can generate both A and *not* A for any proposition A, the system is inconsistent. If the system cannot generate any contradictory pairs of statements it is consistent.

complete A *complete* axiomatic system can derive all true statements by starting from the axioms and following the inference rules. This means if a given proposition is true, some proof for that proposition can be found in the system. Since we do not have a clear definition of *true* (if we defined true as something that can be derived in the system, all axiomatic systems would automatically be complete by definition), we state this more clearly by saying that the system can decide any proposition. This means, for any proposition P, a complete axiomatic system would be able to derive either P or *not* P. A system that cannot decide all statements in the system is *incomplete*. An ideal axiomatic system would be complete and consistent: it would derive all true statements and no false statements.

The completeness of a system depends on the set of possible propositions. Euclid's system is consistent but not complete for the set of propositions about geometry. There are statements that concern simple properties in geometry (a famous example is *any angle can be divided into three equal sub-angles*) that cannot be derived in the system; trisecting an angle requires more powerful tools than the straightedge and compass provided by Euclid's postulates.

Figure 12.1 depicts two axiomatic systems. The one on the left one *incomplete*: there are some propositions that can be stated in the system that are true for which no valid proof exists in the system. The one on the right is *inconsistent*: it is possible to construct valid proofs of both P and *not* P starting from the axioms and following the inference rules. Once a single contradictory proposition can be proven the system becomes completely useless. The contradictory propositions amount to a proof that $true = false$, so once a single pair of contradictory propositions can be proven every other false proposition can also be proven in the system. Hence, only consistent systems are interesting and we focus on whether it is possible for them to also be complete.

Russell's Paradox. Towards the end of the 19th century, many mathematicians sought to systematize mathematics by developing a consistent axiomatic system that is complete for some area of mathematics. One notable attempt was Gottlob Frege's *Grundgestze der Arithmetik* (1893) which attempted to develop an axiomatic system for all of mathematics built from simple logic.



Incomplete system: at least one **true** proposition cannot be proven.

Inconsistent system: there is a valid proof of a proposition that is **false**.

Figure 12.1. Incomplete and inconsistent axiomatic systems.

Bertrand Russell discovered a problem with Frege's system, which is now known as *Russell's paradox*. Suppose R is defined as the set containing all sets that do not contain themselves as members. For example, the set of all prime numbers does not contain itself as a member, so it is a member of R. On the other hand, the set of all entities that are not prime numbers is a member of R. This set contains all sets, since a set is not a prime number, so it must contain itself.

Russell's paradox

The paradoxical question is: *is the set* R *a member of* R*?* There are two possible answers to consider but neither makes sense:

Yes: R is a member of R

We defined the set R as the set of all sets that do not contain themselves as member. Hence, R cannot be a member of itself, and the statement that R is a member of R must be false.

No: R is not a member of R

If R is not a member of R, then R does not contain itself and, by definition, must be a member of set R. This is a contradiction, so the statement that R is not a member of R must be false.

The question is a perfectly clear and precise binary question, but neither the "yes" nor the "no" answer makes any sense. Symbolically, we summarize the paradox: for any set s, $s \in R$ if and only if $s \notin s$. Selecting $s = R$ leads to the contradiction: $R \in R$ if and only if $R \notin R$.

Whitehead and Russell attempted to resolve this paradox by constructing their system to make it impossible to define the set R. Their solution was to introduce types. Each set has an associated type, and a set cannot contain members of its own type. The set types are defined recursively:

- A *type zero set* is a set that contains only non-set objects.
- A *type-n set* can only contain sets of type $n - 1$ and below.

This definition avoids the paradox: the definition of R must now define R as a set of type k set containing all sets of type $k - 1$ and below that do not contain themselves as members. Since R is a type k set, it cannot contain itself, since it cannot contain any type k sets.

Principia Mathematica

In 1913, Whitehead and Russell published *Principia Mathematica*, a bold attempt to mechanize mathematical reasoning that stretched to over 2000 pages. Whitehead and Russell attempted to derive all true mathematical statements about numbers and sets starting from a set of axioms and formal inference rules. They employed the type restriction to eliminate the particular paradox caused by set inclusion, but it does not eliminate all self-referential paradoxes.

For example, consider this paradox named for the Cretan philosopher Epimenides who was purported to have said "All Cretans are liars". If the statement is true, than Epimenides, a Cretan, is not a liar and the statement that all Cretans are liars is false. Another version is the self-referential sentence: *this statement is false*. If the statement is true, then it is true that the statement is false (a contradiction). If the statement is false, then it is a true statement (also a contradiction). It was not clear until Gödel, however, if such statements could be stated in the *Principia Mathematica* system.

12.1.1 Gödel's Incompleteness Theorem

Gödel with Einstein, 1950
Princeton, Institute for Advanced Study Archives

Kurt Gödel was born in Brno (then in Austria-Hungary, now in the Czech Republic) in 1906. Gödel proved that the axiomatic system in *Principia Mathematica* could not be complete and consistent. More generally, Gödel showed that *no* powerful axiomatic system could be both complete and consistent: no matter what the axiomatic system is, if it is powerful enough to express a notion of proof, it must also be the case that there exist statements that can be expressed in the system but cannot be proven either true or false within the system.

Gödel's proof used construction: to prove that *Principia Mathematica* contains statements which cannot be proven either true or false, it is enough to find one such statement. The statement Gödel found:

G_{PM}: Statement G_{PM} does not have any proof in the system of *Principia Mathematica*.

Similarly to Russel's Paradox, this statement leads to a contradiction. It makes no sense for G_{PM} to be either true or false:

Statement G_{PM} is provable in the system.
: If G_{PM} is proven, then it means G_{PM} does have a proof, but G_{PM} stated that G_{PM} has no proof. The system is inconsistent: it can be used to prove a statement that is not true.

Statement G_{PM} is not provable in the system.
: Since G_{PM} cannot be proven in the system, G_{PM} is a true statement. The system is incomplete: we have a true statement that is not provable in the system.

The proof generalizes to *any* axiomatic system, powerful enough to express a corresponding statement G:

G: Statement G does not have any proof in the system.

For the proof to be valid, it is necessary to show that statement G can be expressed in the system.

To express G formally, we need to consider what it means for a statement to not have any proof in the system. A proof of the statement G is a sequence of steps, T_0, T_1, T_2, ..., T_N. Each step is the set of all statements that have been proven

true so far. Initially, T_0 is the set of axioms in the system. To be a proof of G, T_N must contain G. To be a valid proof, each step should be producible from the previous step by applying one of the inference rules to statements from the previous step.

To express statement G an axiomatic system needs to be powerful enough to express the notion that a valid proof does not exist. Gödel showed that such a statement could be constructed using the *Principia Mathematica* system, and using any system powerful enough to be able to express interesting properties. That is, in order for an axiomatic system to be complete and consistent, it must be so weak that it is not possible to express *this statement has no proof* in the system.

12.2 The Halting Problem

Gödel established that no interesting and consistent axiomatic system is capable of proving all true statements in the system. Now we consider the analogous question for computing: *are there problems for which no algorithm exists?*

Recall these definitions form Chapters 1 and 4:

problem: A description of an input and a desired output.

procedure: A specification of a series of actions.

algorithm: A procedure that is guaranteed to always terminate.

A procedure solves a problem if that procedure produces a correct output for every possible input. If that procedure always terminates, it is an algorithm. So, the question can be stated as: *are there problems for which no procedure exists that produces the correct output for every possible problem instance in a finite amount of time?*

A problem is *computable* if there exists an algorithm that solves the problem. It is important to remember that in order for an algorithm to be a solution for a problem P, it must always terminate (otherwise it is not an algorithm) and must always produce the correct output for *all* possible inputs to P. If no such algorithm exists, the problem is *noncomputable*.[1]

computable

noncomputable

Alan Turing proved that noncomputable problems exist. The way to show that uncomputable problems exist is to find one, similarly to the way Gödel showed unprovable true statements exist by finding an unprovable true statement.

The problem Turing found is known as the *Halting Problem*:[2]

Halting Problem

Input: A string representing a Python program.

Output: If evaluating the input program would ever finish, output `True`. Otherwise, output `False`.

[1]The terms *decidable* and *undecidable* are sometimes used to mean the same things as computable and noncomputable.

[2]This problem is a variation on Turing's original problem, which assumed a procedure that takes one input. Of course, Turing did not define the problem using a Python program since Python had not yet been invented when Turing proved the Halting Problem was noncomputable in 1936. In fact, nothing resembling a programmable digital computer would emerge until several years later.

Suppose we had a procedure *halts* that solves the Halting Problem. The input to *halts* is a Python program expressed as a string.

For example, *halts*('(+ 2 3)') should evaluate to True, *halts*('while True: pass') should evaluate to False (the Python **pass** statement does nothing, but is needed to make the while loop syntactically correct), and

```
halts('''
  def fibo(n):
   if n == 1 or n == 2: return 1
   else: return fibo(n−1) + fibo(n−2)
  fibo(60)
 ''')
```

should evaluate to True. From the last example, it is clear that *halts* cannot be implemented by evaluating the expression and outputting True if it terminates. The problem is knowing when to give up and output False. As we analyzed in Chapter 7, evaluating *fibo*(60) would take trillions of years; in theory, though, it eventually finishes so *halts* should output True.

This argument is not sufficient to prove that *halts* is noncomputable. It just shows that one particular way of implementing *halts* would not work. To show that *halts* is noncomputable, we need to show that it is impossible to implement a *halts* procedure that would produce the correct output for all inputs in a finite amount of time.

Here is another example that suggests (but does not prove) the impossibility of *halts* (where *sumOfTwoPrimes* is defined as an algorithm that take a number as input and outputs True if the number is the sum of two prime numbers and False otherwise):

```
halts('n = 4; while sumOfTwoPrimes(n): n = n + 2')
```

This program halts if there exists an even number greater than 2 that is not the sum of two primes. We assume unbounded integers even though every actual computer has a limit on the largest number it can represent. Our computing model, though, uses an infinite tape, so there is no arbitrary limit on number sizes.

Knowing whether or not the program halts would settle an open problem known as Goldbach's Conjecture: *every even integer greater than 2 can be written as the sum of two primes.* Christian Goldbach proposed a form of the conjecture in a letter to Leonhard Euler in 1742. Euler refined it and believed it to be true, but couldn't prove it.

With a *halts* algorithm, we could settle the conjecture using the expression above: if the result is False, the conjecture is proven; if the result is True, the conjecture is disproved. We could use a *halts* algorithm like this to resolve many other open problems. This strongly suggests there is no *halts* algorithm, but does not prove it cannot exist.

Proving Noncomputability. Proving non-existence is requires more than just showing a hard problem could be solved if something exists. One way to prove non-existence of an X, is to show that if an X exists it leads to a contradiction.

We prove that the existence of a *halts* algorithm leads to a contradiction, so no *halts* algorithm exists.

We obtain the contradiction by showing one input for which the *halts* procedure could not possibly work correctly. Consider this procedure:

```
def paradox():
  if halts('paradox()'): while True: pass
```

The body of the *paradox* procedure is an if expression. The consequent expression is a never-ending loop.

The predicate expression cannot sensibly evaluate to either True or False:

halts('paradox()') ⇒ True
: If the predicate expression evaluates to True, the consequent block is evaluated producing a never-ending loop. Thus, if *halts*('paradox()') evaluates to True, the evaluation of an application of *paradox* never halts. But, this means the result of *halts*('paradox()') was incorrect.

halts('paradox()') ⇒ False
: If the predicate expression evaluates to False, the alternate block is evaluated. It is empty, so evaluation terminates. Thus, the evaluation of *paradox*() terminates, contradicting the result of *halts*('paradox()').

Either result for *halts*(`*paradox*()') leads to a contradiction! The only sensible thing *halts* could do for this input is to not produce a value. That means there is no algorithm that solves the Halting Problem. Any procedure we define to implement *halts* must sometimes either produce the wrong result or fail to produce a result at all (that is, run forever without producing a result). This means the Halting Problem is noncomputable.

There is one important hole in our proof: we argued that because *paradox* does not make sense, something in the definition of *paradox* must not exist and identified *halts* as the component that does not exist. This assumes that everything else we used to define *paradox* does exist.

This seems reasonable enough—they are built-in to Python so they seem to exist. But, perhaps the reason *paradox* leads to a contradiction is because True does not really exist or because it is not possible to implement an if expression that strictly follows the Python evaluation rules. Although we have been using these and they seems to always work fine, we have no formal model in which to argue that evaluating True always terminates or that an if expression means exactly what the evaluation rules say it does.

Our informal proof is also insufficient to prove the stronger claim that no algorithm exists to solve the halting problem. All we have shown is that no Python procedure exists that solves *halts*. Perhaps there is a procedure in some more powerful programming language in which it is possible to implement a solution to the Halting Problem. In fact, we will see that no more powerful programming language exists.

A convincing proof requires a formal model of computing. This is why Alan Turing developed a model of computation.

12.3 Universality

Recall the Turing Machine model from Chapter 6: a Turing Machine consists of an infinite tape divided into discrete square into which symbols from a fixed alphabet can be written, and a tape head that moves along the tape. On each step, the tape head can read the symbol in the current square, write a symbol in the current square, and move left or right one square or halt. The machine can keep track of a finite number of possible states, and determines which action to take based on a set of transition rules that specify the output symbol and head action for a given current state and read symbol.

Turing argued that this simple model corresponds to our intuition about what can be done using mechanical computation. Recall this was 1936, so the model for mechanical computation was not what a mechanical computer can do, but what a human computer can do. Turing argued that his model corresponded to what a human computer could do by following a systematic procedure: the infinite tape was as powerful as a two-dimensional sheet of paper or any other recording medium, the set of symbols must be finite otherwise it would not be possible to correctly distinguish all symbols, and the number of machine states must be finite because there is a limited amount a human can keep in mind at one time.

We can enumerate all possible Turing Machines. One way to see this is to devise a notation for writing down any Turing Machine. A Turing Machine is completely described by its alphabet, states and transition rules. We could write down any Turing Machine by numbering each state and listing each transition rule as a tuple of the current state, alphabet symbol, next state, output symbol, and tape direction. We can map each state and alphabet symbol to a number, and use this encoding to write down a unique number for every possible Turing Machine. Hence, we can enumerate all possible Turing Machines by just enumerating the positive integers. Most positive integers do not correspond to valid Turing Machines, but if we go through all the numbers we will eventually reach every possible Turing Machine.

This is step towards proving that some problems cannot be solved by any algorithm. The number of Turing Machines is less than the number of real numbers. Both numbers are infinite, but as explained in Section 1.2.2, Cantor's diagonalization proof showed that the real numbers are not countable. Any attempt to map the real numbers to the integers must fail to include all the real numbers. This means there are real numbers that cannot be produced by any Turing Machine: there are fewer Turing Machines than there are real numbers, so there must be some real numbers that cannot be produced by any Turing Machine.

Universal Turing Machine

The next step is to define the machine depicted in Figure 12.2. A *Universal Turing Machine* is a machine that takes as input a number that identifies a Turing Machine and simulates the specified Turing Machine running on initially empty input tape.

The Universal Turing Machine can simulate any Turing Machine. In his proof, Turing describes the transition rules for such a machine. It simulates the Turing Machine encoded by the input number. One can imagine doing this by using the tape to keep track of the state of the simulated machine. For each step, the universal machine searches the description of the input machine to find the ap-

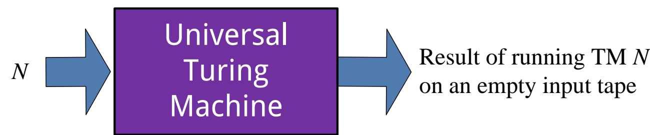


Figure 12.2. Universal Turing Machine.

propriate rule. This is the rule for the current state of the simulated machine on the current input symbol of the simulated machine. The universal machine keeps track of the machine and tape state of the simulated machine, and simulates each step. Thus, there is a single Turing Machine that can simulate every Turing Machine.

Since a Universal Turing Machine can simulate every Turing Machine, and a Turing Machine can perform any computation according to our intuitive notion of computation, this means a Universal Turing Machine can perform all computations. Using the universal machine and a diagonalization argument similar to the one above for the real numbers, Turing reached a similar contradiction for a problem analogous to the Halting Problem for Python programs but for Turing Machines instead.

If we can simulate a Universal Turing Machine in a programming language, that language is a *universal programming language*. There is some program that can be written in that language to perform every possible computation.

universal programming language

To show that a programming language is universal, it is sufficient to show that it can simulate any Turing Machine, since a Turing Machine can perform every possible computation. To simulate a Universal Turing Machine, we need some way to keep track of the state of the tape (for example, the list datatypes in Scheme or Python would be adequate), a way to keep track of the internal machine state (a number can do this), and a way to execute the transition rules (we could define a procedure that does this using an if expression to make decisions about which transition rule to follow for each step), and a way to keep going (we can do this in Scheme with recursive applications). Thus, Scheme is a universal programming language: one can write a Scheme program to simulate a Universal Turing Machine, and thus, perform any mechanical computation.

12.4 Proving Non-Computability

We can show that a problem is computable by describing a procedure and proving that the procedure always terminates and always produces the correct answer. It is enough to provide a convincing argument that such a procedure exists; finding the actual procedure is not necessary (but often helps to make the argument more convincing).

To show that a problem is not computable, we need to show that *no* algorithm exists that solves the problem. Since there are an infinite number of possible procedures, we cannot just list all possible procedures and show why each one does not solve the problem. Instead, we need to construct an argument showing that if there were such an algorithm it would lead to a contradiction.

The core of our argument is based on knowing the Halting Problem is noncomputable. If a solution to some new problem P could be used to solve the Halting

Problem, then we know that P is also noncomputable. That is, no algorithm exists that can solve P since if such an algorithm exists it could be used to also solve the Halting Problem which we already know is impossible.

Reduction Proofs. The proof technique where we show that a solution for some problem P can be used to solve a different problem Q is known as a *reduction*.

reduction

reducible

A problem Q is *reducible* to a problem P if a solution to P could be used to solve Q. This means that problem Q is no harder than problem P, since a solution to problem Q leads to a solution to problem P.

Example 12.1: Prints-Three Problem

Consider the problem of determining if an application of a procedure would ever print 3:

Prints-Three

Input: A string representing a Python program.

Output: If evaluating the input program would print 3, output True; otherwise, output False.

We show the Prints-Three Problem is noncomputable by showing that it is as hard as the Halting Problem, which we already know is noncomputable.

Suppose we had an algorithm *printsThree* that solves the Prints-Three Problem. Then, we could define *halts* as:

```
def halts(p):
  return printsThree(p + '; print(3)')
```

The *printsThree* application would evaluate to True if evaluating the Python program specified by p would halt since that means the **print**(3) statement appended to p would be evaluated. On the other hand, if evaluating p would not halt, the added print statement never evaluated. As long as the program specified by p would never print 3, the application of *printsThree* should evaluate to False. Hence, if a *printsThree* algorithm exists, we would use it to implement an algorithm that solves the Halting Problem.

The one wrinkle is that the specified input program might print 3 itself. We can avoid this problem by transforming the input program so it would never print 3 itself, without otherwise altering its behavior. One way to do this would be to replace all occurrences of **print** (or any other built-in procedure that prints) in the string with a new procedure, *dontprint* that behaves like **print** but doesn't actually print out anything. Suppose the *replacePrints* procedure is defined to do this. Then, we could use *printsThree* to define *halts*:

```
def halts(p): return printsThree(replacePrints(p) + '; print(3)')
```

We know that the Halting Problem is noncomputable, so this means the Prints-Three Problem must also be noncomputable.

Exploration 12.1: Virus Detection

The Halting Problem and Prints-Three Problem are noncomputable, but do seem to be obviously important problems. It is useful to know if a procedure application will terminate in a reasonable amount of time, but the Halting Problem

does not answer that question. It concerns the question of whether the procedure application will terminate in any finite amount of time, no matter how long it is. This example considers a problem for which it would be very useful to have a solution for it one existed.

A virus is a program that infects other programs. A virus spreads by copying its own code into the code of other programs, so when those programs are executed the virus will execute. In this manner, the virus spreads to infect more and more programs. A typical virus also includes a malicious payload so when it executes in addition to infecting other programs it also performs some damaging (corrupting data files) or annoying (popping up messages) behavior. The Is-Virus Problem is to determine if a procedure specification contains a virus:

Is-Virus

Input: A specification of a Python program.

Output: If the expression contains a virus (a code fragment that will infect other files) output True. Otherwise, output False.

We demonstrate the Is-Virus Problem is noncomputable using a similar strategy to the one we used for the Prints-Three Problem: we show how to define a *halts* algorithm given a hypothetical *isVirus* algorithm. Since we know *halts* is noncomputable, this shows there is no *isVirus* algorithm.

Assume *infectFiles* is a procedure that infects files, so the result of evaluating *isVirus*('infectFiles()') is True. We could define *halts* as:

```
def halts(p):
  return isVirus(p + '; infectFiles()')
```

This works as long as the program specified by *p* does not exhibit the file-infecting behavior. If it does, *p* could infect a file and never terminate, and *halts* would produce the wrong output. To solve this we need to do something like we did in the previous example to hide the printing behavior of the original program.

A rough definition of file-infecting behavior would be to consider any write to an executable file to be an infection. To avoid any file infections in the specific program, we replace all procedures that write to files with procedures that write to shadow copies of these files. For example, we could do this by creating a new temporary directory and prepend that path to all file names. We call this (assumed) procedure, *sandBox*, since it transforms the original program specification into one that would execute in a protected sandbox.

```
def halts(p): isVirus(sandBox(p) + '; infectFiles()')
```

Since we know there is no algorithm that solves the Halting Problem, this proves that there is no algorithm that solves the Is-Virus problem.

Virus scanners such as Symantec's Norton AntiVirus attempt to solve the Is-Virus Problem, but its non-computability means they are doomed to always fail. Virus scanners detect known viruses by scanning files for strings that match signatures in a database of known viruses. As long as the signature database is frequently updated they may be able to detect currently spreading viruses, but this approach cannot detect a new virus that will not match the signature of a previously known virus.

Sophisticated virus scanners employ more advanced techniques to attempt to detect complex viruses such as metamorphic viruses that alter their own code as they propagate to avoid detection. But, because the general Is-Virus Problem is noncomputable, we know that it is impossible to create a program that always terminates and that always correctly determines if an input procedure specification is a virus.

I am rather puzzled why you draw this distinction between proof finders and proof checkers. It seems to me rather unimportant as one can always get a proof finder from a proof checker, and the converse is almost true: the converse false if for instance one allows the proof finder to go through a proof in the ordinary way, and then, rejecting the steps, to write down the final formula as a 'proof' of itself. One can easily think up suitable restrictions on the idea of proof which will make this converse true and which agree well with our ideas of what a proof should be like. I am afraid this may be more confusing to you than enlightening.
Alan Turing, letter to Max Newman, 1940

Exercise 12.1. Is the Launches-Missiles Problem described below computable? Provide a convincing argument supporting your answer.

Launches-Missiles

Input: A specification of a procedure.

Output: If an application of the procedure would lead to the missiles being launched, outputs True. Otherwise, outputs False.

You may assume that the only thing that causes the missiles to be launched is an application of the *launchMissiles* procedure.

Exercise 12.2. Is the Same-Result Problem described below computable? Provide a convincing argument supporting your answer.

Same-Result

Input: Specifications of two procedures, P and Q.

Output: If an application of P terminates and produces the same value as applying Q, outputs True. If an application of P does not terminate, and an application of Q also does not terminate, outputs True. Otherwise, outputs False.

Exercise 12.3. Is the Check-Proof Problem described below computable? Provide a convincing argument supporting your answer.

Check-Proof

Input: A specification of an axiomatic system, a statement (the theorem), and a proof (a sequence of steps, each identifying the axiom that is applied).

Output: Outputs True if the proof is a valid proof of the theorem in the system, or False if it is not a valid proof.

Exercise 12.4. Is the Find-Finite-Proof Problem described below computable? Provide a convincing argument supporting your answer.

Find-Finite-Proof

Input: A specification of an axiomatic system, a statement (the theorem), and a maximum number of steps (max-steps).

Output: If there is a proof in the axiomatic system of the theorem that uses max-steps or fewer steps, outputs True. Otherwise, outputs False.

Exercise 12.5. [⋆] Is the Find-Proof Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Find-Proof

Input: A specification of an axiomatic system, and a statement (the theorem).

Output: If there is a proof in the axiomatic system of the theorem, outputs True. Otherwise, outputs False.

Exploration 12.2: Busy Beavers

Consider the Busy-Beaver Problem (devised by Tibor Radó in 1962):

Busy-Beaver

Input: A positive integer, n.

Output: A number representing that maximum number of steps a Turing Machine with n states and a two-symbol tape alphabet can run starting on an empty tape before halting.

We use 0 and 1 for the two tape symbols, where the blank squares on the tape are interpreted as 0s (alternately, we could use *blank* and X as the symbols, but it is more natural to describe machines where symbols are 0 and 1, so we can think of the initially blank tape as containing all 0s).

For example, if the Busy Beaver input n is 1, the output should be 1. The best we can do with only one state is to halt on the first step. If the transition rule for a 0 input moves left, then it will reach another 0 square and continue forever without halting; similarly it if moves right.

For $n = 2$, there are more options to consider. The machine in Figure 12.3 runs for 6 steps before halting, and there is no two-state machine that runs for more steps. One way to support this claim would be to try simulating all possible two-state Turing Machines.

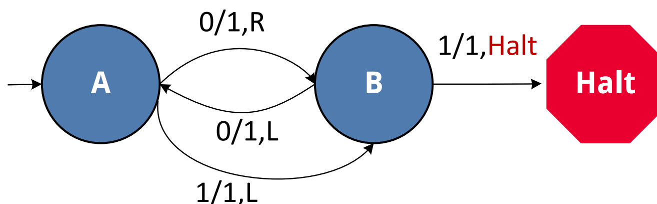


Figure 12.3. Two-state Busy Beaver Machine.

Busy Beaver numbers increase extremely quickly. The maximum number of steps for a three-state machine is 21, and for a four-state machine is 107. The value for a five-state machine is not yet known, but the best machine found to date runs for 47,176,870 steps! For six states, the best known result, discovered in 2007 by Terry Ligocki and Shawn Ligocki, is over 2879 decimal digits long.

We can prove the Busy Beaver Problem is noncomputable by reducing the Halting Problem to it. Suppose we had an algorithm, $bb(n)$, that takes the number of states as input and outputs the corresponding Busy Beaver. Then, we could solve the Halting Problem for a Turing Machine:

TM Halting Problem

Input: A string representing a Turing Machine.

Output: If executing the input Turing Machine starting with a blank tape would ever finish, output True. Otherwise, output False.

The TM Halting Problem is different from the Halting Problem as we defined it earlier, so first we need to show that the TM Halting Problem is noncomputable by showing it could be used to solve the Python Halting Problem. Because Python is universal programming language, it is possible to transform any Turing Machine into a Python program. Once way to do this would be to write a Universal Turing Machine simulator in Python, and then create a Python program that first creates a tape containing the input Turing Machine description, and then calls the Universal Turing Machine simulator on that input. This shows that the TM Halting Problem is noncomputable.

Next, we show that an algorithm that solves the Busy Beaver Problem could be used to solve the TM Halting Problem. Here's how (in Pythonish pseudocode):

```
def haltsTM(m):
  states = numberOfStates(m)
  maxSteps = bb(states)
  state = 0
  tape = []
  for step in range(0, maxSteps):
    state, tape = simulateOneStep(m, state, tape)
    if halted(state): return True
  return False
```

The *simulateOneStep* procedure takes as inputs a Turing Machine description, its current state and tape, and simulates the next step on the machine. So, *haltsTM* simulates up to $bb(n)$ steps of the input machine m where n is the number of states in m. Since $bb(n)$ is the maximum number of steps a Turing Machine with n states can execute before halting, we know if m has not halted in the simulate before *maxSteps* is reached that the machine m will never halt, and can correctly return False. This means there is no algorithm that can solve the Busy Beaver Problem.

Exercise 12.6. Confirm that the machine showing in Figure 12.3 runs for 6 steps before halting.

Exercise 12.7. Prove the Beaver Bound problem described below is also noncomputable:

Beaver-Bound

Input: A positive integer, n.

Output: A number that is greater than the maximum number of steps a Turing Machine with n states and a two-symbol tape alphabet can run starting on an empty tape before halting.

A valid solution to the Beaver-Bound problem can produce any result for n as long as it is greater than the Busy Beaver value for n.

Exercise 12.8. [⋆⋆⋆] Find a 5-state Turing Machine that runs for more than 47,176,870 steps, or prove that no such machine exists.

12.5 Summary

Although today's computers can do amazing things, many of which could not even have been imagined twenty years ago, there are problems that can never be solved by computing. The Halting Problem is the most famous example: it is impossible to define a mechanical procedure that always terminates and correctly determines if the computation specified by its input would terminate. Once we know the Halting Problem is noncomputable, we can show that other problems are also noncomputable by illustrating how a solution to the other problem could be used to solve the Halting Problem which we know to be impossible.

Noncomputable problems frequently arise in practice. For example, identifying viruses, analyzing program paths, and constructing proofs, are all noncomputable problems.

Just because a problem is noncomputable does not mean we cannot produce useful programs that address the problem. These programs provide approximate solutions, which are often useful in practice. They produce the correct results on many inputs, but on some inputs must either fail to produce any result or produce an incorrect result.