

Learning Objectives:

- More examples of uncomputable functions that are not as tied to computation.
- Gödel's incompleteness theorem - a result that shook the world of mathematics in the early 20th century.

10

Is every theorem provable?

"Take any definite unsolved problem, such as ... the existence of an infinite number of prime numbers of the form $2^n + 1$. 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.

"The meaning of a statement is its method of verification.", Moritz Schlick, 1938 (aka "The verification principle" of logical positivism)

The problems shown uncomputable in [Chapter 8](#), while natural and important, still intimately involved NAND-TM programs or other computing mechanisms in their definitions. One could perhaps hope that as long as we steer clear of functions whose inputs are themselves programs, we can avoid the "curse of uncomputability". Alas, we have no such luck.

In this chapter we will see an example of a natural and seemingly "computation free" problem that nevertheless turns out to be uncomputable: solving Diophantine equations. As a corollary, we will see one of the most striking results of 20th century mathematics: *Gödel's Incompleteness Theorem*, which showed that there are some mathematical statements (in fact, in number theory) that are *inherently unprovable*. We will actually start with the latter result, and then show the former.

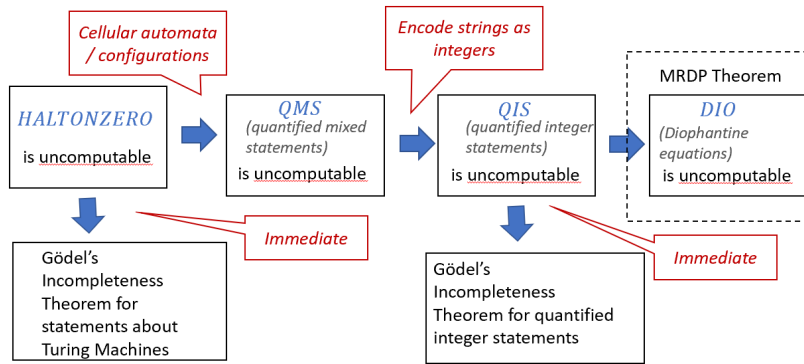


Figure 10.1: Outline of the results of this chapter. One version of Gödel's Incompleteness Theorem is an immediate consequence of the uncomputability of the Halting problem. To obtain the theorem as originally stated (for statements about the integers) we first prove that the QMS problem of determining truth of quantified statements involving both integers and strings is uncomputable. We do so using the notion of *Turing Machine configurations* but there are alternative approaches to do so as well, see [Remark 10.14](#).

10.1 HILBERT'S PROGRAM AND GÖDEL'S INCOMPLETENESS THEOREM

"And what are these ...vanishing increments? They are neither finite quantities, nor quantities infinitely small, nor yet nothing. May we not call them the ghosts of departed quantities?", George Berkeley, Bishop of Cloyne, 1734.

The 1700's and 1800's were a time of great discoveries in mathematics but also of several crises. The discovery of calculus by Newton and Leibnitz in the late 1600's ushered a golden age of problem solving. Many longstanding challenges succumbed to the new tools that were discovered, and mathematicians got ever better at doing some truly impressive calculations. However, the rigorous foundations behind these calculations left much to be desired. Mathematicians manipulated infinitesimal quantities and infinite series cavalierly, and while most of the time they ended up with the correct results, there were a few strange examples (such as trying to calculate the value of the infinite series $1 - 1 + 1 - 1 + 1 + \dots$) which seemed to give out different answers depending on the method of calculation. This led to a growing sense of unease in the foundations of the subject which was addressed in works of mathematicians such as Cauchy, Weierstrass, and Riemann, who eventually placed analysis on firmer foundations, giving rise to the ϵ 's and δ 's that students taking honors calculus grapple with to this day.

In the beginning of the 20th century, there was an effort to replicate this effort, in greater rigor, to all parts of mathematics. The hope was to show that all the true results of mathematics can be obtained by starting with a number of axioms, and deriving theorems from them

using logical rules of inference. This effort was known as the *Hilbert program*, named after the influential mathematician David Hilbert.

Alas, it turns out the results we've seen dealt a devastating blow to this program, as was shown by Kurt Gödel in 1931:

Theorem 10.1 — Gödel's Incompleteness Theorem: informal version. For every sound proof system V for sufficiently rich mathematical statements, there is a mathematical statement that is *true* but is not *provable* in V .

10.1.1 Defining “Proof Systems”

Before proving [Theorem 10.3](#), we need to define “proof systems” and even formally define the notion of a “mathematical statement”. In geometry and other areas of mathematics, proof systems are often defined by starting with some basic assumptions or *axioms* and then deriving more statements by using *inference rules* such as the famous *Modus Ponens*, but what axioms shall we use? What rules? We will use an extremely general notion of proof systems, not even restricting ourselves to ones that have the form of axioms and inference.

Mathematical statements. At the highest level, a mathematical statement is simply a piece of text, which we can think of as a *string* $x \in \{0, 1\}^*$. Mathematical statements contain assertions whose truth does not depend on any empirical fact, but rather only on properties of abstract objects. For example, the following is a mathematical statement:¹

“The number 2,696,635,869,504,783,333,238,805,675,613, 588,278,597,832,162,617,892,474,670,798,113 is prime”.

Mathematical statements do not have to involve numbers. They can assert properties of any other mathematical object including sets, strings, functions, graphs and yes, even *programs*. Thus, another example of a mathematical statement is the following:²

The following Python function halts on every positive integer n

```
def f(n):
    if n==1: return 1
    return f(3*n+1) if n % 2 else f(n//2)
```

Proof systems. A *proof* for a statement $x \in \{0, 1\}^*$ is another piece of text $w \in \{0, 1\}^*$ that certifies the truth of the statement asserted in x . The conditions for a valid proof system are:

¹ This happens to be a *false* statement.

² It is *unknown* whether this statement is true or false.

1. (*Effectiveness*) Given a statement x and a proof w , there is an algorithm to verify whether or not w is a valid proof for x . (For example, by going line by line and checking that each line follows from the preceding ones using one of the allowed inference rules.)
2. (*Soundness*) If there is a valid proof w for x then x is true.

These are quite minimal requirements for a proof system. Requirement 2 (soundness) is the very definition of a proof system: you shouldn't be able to prove things that are not true. Requirement 1 is also essential. If there is no set of rules (i.e., an algorithm) to check that a proof is valid then in what sense is it a proof system? We could replace it with the system where the "proof" for a statement x is "trust me: it's true".

We formally define proof systems as an algorithm V where $V(x, w) = 1$ holds if the string w is a valid proof for the statement x . Even if x is true, the string w does not have to be a valid proof for it (there are plenty of wrong proofs for true statements such as $4=2+2$) but if w is a valid proof for x then x must be true.

Definition 10.2 — Proof systems. Let $\mathcal{T} \subseteq \{0, 1\}^*$ be some set (which we consider the "true" statements). A *proof system* for \mathcal{T} is an algorithm V that satisfies:

1. (*Effectiveness*) For every $x, w \in \{0, 1\}^*$, $V(x, w)$ halts with an output of either 0 or 1.
2. (*Soundness*) For every $x \notin \mathcal{T}$ and $w \in \{0, 1\}^*$, $V(x, w) = 0$.

A true statement $x \in \mathcal{T}$ is *unprovable* (with respect to V) if for every $w \in \{0, 1\}^*$, $V(x, w) = 0$. We say that V is *complete* if there does not exist a true statement x that is unprovable with respect to v .

💡 Big Idea 14 A *proof* is just a string of text whose meaning is given by a *verification algorithm*.

10.2 GÖDEL'S INCOMPLETENESS THEOREM: COMPUTATIONAL VARIANT

Our first formalization of [Theorem 10.3](#) involves statements about Turing machines. We let \mathcal{H} be the set of strings $x \in \{0, 1\}^*$ that have the form "Turing machine M halts on the zero input".

Theorem 10.3 — Gödel's Incompleteness Theorem: computational variant.

There does not exist a complete proof system for \mathcal{H} .

Proof Idea:

If we had such a complete and sound proof system then we could solve the *HALTONZERO* problem. On input a Turing machine M , we would search all purported proofs w and halt as soon as we find a proof of either “ M halts on zero” or “ M does not halt on zero”. If the system is sound and complete then we will eventually find such a proof, and it will provide us with the correct output.

★

Proof of Theorem 10.3. Assume for the sake of contradiction that there was such a proof system V . We will use V to build an algorithm A that computes *HALTONZERO*, hence contradicting Theorem 8.10. Our algorithm A will work as follows:

Algorithm 10.4 — Halting from proofs.

Input: Turing Machine M

Output: 1 if M halts on the

Input: 0; 0 otherwise.

```

1: for  $n = 1, 2, 3, \dots$  do
2:   for  $w \in \{0, 1\}^n$  do
3:     if  $V(\text{"}M \text{ halts on } 0\text{"}, w) = 1$  then
4:       return 1
5:     end if
6:     if  $V(\text{"}M \text{ does not halt on } 0\text{"}, w) = 1$  then
7:       return 0
8:     end if
9:   end for
10: end for
```

If M halts on 0 then under our assumption there exists w that proves this fact, and so when Algorithm A reaches $n = |w|$ we will eventually find this w and output 1, unless we already halted before. But we cannot halt before and output a wrong answer because it would contradict the soundness of the proof system. Similarly, this shows that if M does *not* halt on 0 then (since we assume there is a proof of this fact too) our algorithm A will eventually halt and output 0.

■

R

Remark 10.5 — The Gödel statement (optional). One can extract from the proof of [Theorem 10.3](#) a procedure that for every proof system V , yields a true statement x^* that cannot be proven in V . But Gödel's proof gave a very explicit description of such a statement x^* which is closely related to the “[Liar's paradox](#)”. That is, Gödel's statement x^* was designed to be true if and only if $\forall_{w \in \{0,1\}^*} V(x, w) = 0$. In other words, it satisfied the following property

$$x^* \text{ is true} \Leftrightarrow x^* \text{ does not have a proof in } V \quad (10.1)$$

One can see that if x^* is true, then it does not have a proof, but it is false then (assuming the proof system is sound) then it cannot have a proof, and hence x^* must be both true and unprovable. One might wonder how is it possible to come up with an x^* that satisfies a condition such as (10.1) where the same string x^* appears on both the righthand side and the lefthand side of the equation. The idea is that the proof of [Theorem 10.3](#) yields a way to transform every statement x into a statement $F(x)$ that is true if and only if x does not have a proof in V . Thus x^* needs to be a *fixed point* of F : a sentence such that $x^* = F(x^*)$. It turns out that [we can always find](#) such a fixed point of F . We've already seen this phenomenon in the λ calculus, where the Y combinator maps every F into a fixed point YF of F . This is very related to the idea of programs that can print their own code. Indeed, Scott Aaronson likes to describe Gödel's statement as follows:

The following sentence repeated twice, the second time in quotes, is not provable in the formal system V . “The following sentence repeated twice, the second time in quotes, is not provable in the formal system V .”

In the argument above we actually showed that x^* is *true*, under the assumption that V is sound. Since x^* is true and does not have a proof in V , this means that we cannot carry the above argument in the system V , which means that V cannot prove its own soundness (or even consistency: that there is no proof of both a statement and its negation). Using this idea, it's not hard to get Gödel's second incompleteness theorem, which says that every sufficiently rich V cannot prove its own consistency. That is, if we formalize the statement c^* that is true if and only if V is consistent (i.e.,

V cannot prove both a statement and the statement's negation), then c^* cannot be proven in V .

10.3 QUANTIFIED INTEGER STATEMENTS

There is something “unsatisfying” about [Theorem 10.3](#). Sure, it shows there are statements that are unprovable, but they don't feel like “real” statements about math. After all, they talk about *programs* rather than numbers, matrices, or derivatives, or whatever it is they teach in math courses. It turns out that we can get an analogous result for statements such as “there are no positive integers x and y such that $x^2 - 2 = y^7$ ”, or “there are positive integers x, y, z such that $x^2 + y^6 = z^{11}$ ” that only talk about *natural numbers*. It doesn't get much more “real math” than this. Indeed, the 19th century mathematician Leopold Kronecker famously said that “God made the integers, all else is the work of man.” (By the way, the status of the above two statements is [unknown](#).)

To make this more precise, let us define the notion of *quantified integer statements*:

Definition 10.6 — Quantified integer statements. A *quantified integer statement* is a well-formed statement with no unbound variables involving integers, variables, the operators $>, <, \times, +, -, =$, the logical operations \neg (NOT), \wedge (AND), and \vee (OR), as well as quantifiers of the form $\exists_{x \in \mathbb{N}}$ and $\forall_{y \in \mathbb{N}}$ where x, y are variable names.

We often care deeply about determining the truth of quantified integer statements. For example, the statement that [Fermat's Last Theorem](#) is true for $n = 3$ can be phrased as the quantified integer statement

$$\neg \exists_{a \in \mathbb{N}} \exists_{b \in \mathbb{N}} \exists_{c \in \mathbb{N}} (a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (a \times a \times a + b \times b \times b = c \times c \times c) . \quad (10.2)$$

The [twin prime conjecture](#), that states that there is an infinite number of numbers p such that both p and $p + 2$ are primes can be phrased as the quantified integer statement

$$\forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \wedge \text{PRIME}(p) \wedge \text{PRIME}(p + 2) \quad (10.3)$$

where we replace an instance of $\text{PRIME}(q)$ with the statement $(q > 1) \wedge \forall_{a \in \mathbb{N}} \forall_{b \in \mathbb{N}} (a = 1) \vee (a = q) \vee \neg(a \times b = q)$.

The claim (mentioned in Hilbert's quote above) that there are infinitely many primes of the form $p = 2^n + 1$ can be phrased as follows:

$$\begin{aligned} & \forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \wedge \text{PRIME}(p) \wedge \\ & (\forall_{k \in \mathbb{N}} (k \neq 2 \wedge \text{PRIME}(k) \Rightarrow \neg \text{DIVIDES}(k, p - 1)) \end{aligned} \quad (10.4)$$

where $\text{DIVIDES}(a, b)$ is the statement $\exists_{c \in \mathbb{N}} b \times c = a$. In English, this corresponds to the claim that for every n there is some $p > n$ such that all of $p - 1$'s prime factors are equal to 2.



Remark 10.7 — Syntactic sugar for quantified integer statements. To make our statements more readable, we often use syntactic sugar and so write $x \neq y$ as shorthand for $\neg(x = y)$, and so on. Similarly, the “implication operator” $a \Rightarrow b$ is “syntactic sugar” or shorthand for $\neg a \vee b$, and the “if and only if operator” $a \Leftrightarrow b$ is shorthand for $(a \Rightarrow b) \wedge (b \Rightarrow a)$. We will also allow ourselves the use of “macros”: plugging in one quantified integer statement in another, as we did with DIVIDES and PRIME above.

Much of number theory is concerned with determining the truth of quantified integer statements. Since our experience has been that, given enough time (which could sometimes be several centuries) humanity has managed to do so for the statements that it cared enough about, one could (as Hilbert did) hope that eventually we would be able to prove or disprove all such statements. Alas, this turns out to be impossible:

Theorem 10.8 — Gödel's Incompleteness Theorem for quantified integer statements. Let $V : \{0, 1\}^* \rightarrow \{0, 1\}$ a computable purported verification procedure for quantified integer statements. Then either:

- *V is not sound:* There exists a false statement x and a string $w \in \{0, 1\}^*$ such that $V(x, w) = 1$.
- or*
- *V is not complete:* There exists a true statement x such that for every $w \in \{0, 1\}^*$, $V(x, w) = 0$.

Theorem 10.8 is a direct corollary of the following result, just as Theorem 10.3 was a direct corollary of the uncomputability of HALTONZERO :

Theorem 10.9 — Uncomputability of quantified integer statements. Let $QIS : \{0, 1\}^* \rightarrow \{0, 1\}$ be the function that given a (string representation of) a quantified integer statement outputs 1 if it is true and 0 if it is false. Then QIS is uncomputable.

Since a quantified integer statement is simply a sequence of symbols, we can easily represent it as a string. For simplicity we will assume that *every* string represents some quantified integer statement, by mapping strings that do not correspond to such a statement to an arbitrary statement such as $\exists_{x \in \mathbb{N}} x = 1$.

P

Please stop here and make sure you understand why the uncomputability of QIS (i.e., Theorem 10.9) means that there is no sound and complete proof system for proving quantified integer statements (i.e., Theorem 10.8). This follows in the same way that Theorem 10.3 followed from the uncomputability of *HALTONZERO*, but working out the details is a great exercise (see Exercise 10.1)

In the rest of this chapter, we will show the proof of Theorem 10.8, following the outline illustrated in ??.

10.4 DIOPHANTINE EQUATIONS AND THE MRDP THEOREM

Many of the functions people wanted to compute over the years involved solving equations. These have a much longer history than mechanical computers. The Babylonians already knew how to solve some quadratic equations in 2000BC, and the formula for all quadratics appears in the *Bakhshali Manuscript* that was composed in India around the 3rd century. During the Renaissance, Italian mathematicians discovered generalization of these formulas for cubic and quartic (degrees 3 and 4) equations. Many of the greatest minds of the 17th and 18th century, including Euler, Lagrange, Leibniz and Gauss worked on the problem of finding such a formula for *quintic* equations to no avail, until in the 19th century Ruffini, Abel and Galois showed that no such formula exists, along the way giving birth to *group theory*.

However, the fact that there is no closed-form formula does not mean we can not solve such equations. People have been solving higher degree equations numerically for ages. The Chinese manuscript *Jiuzhang Suanshu* from the first century mentions such approaches. Solving polynomial equations is by no means restricted only to ancient history or to students' homeworks. The *gradient descent* method is the workhorse powering many of the machine

learning tools that have revolutionized Computer Science over the last several years.

But there are some equations that we simply do not know how to solve *by any means*. For example, it took more than 200 years until people succeeded in proving that the equation $a^{11} + b^{11} = c^{11}$ has no solution in integers.³ The notorious difficulty of so called *Diophantine equations* (i.e., finding *integer* roots of a polynomial) motivated the mathematician David Hilbert in 1900 to include the question of finding a general procedure for solving such equations in his famous list of twenty-three open problems for mathematics of the 20th century. I don't think Hilbert doubted that such a procedure exists. After all, the whole history of mathematics up to this point involved the discovery of ever more powerful methods, and even impossibility results such as the inability to trisect an angle with a straightedge and compass, or the non-existence of an algebraic formula for quintic equations, merely pointed out to the need to use more general methods.

Alas, this turned out not to be the case for Diophantine equations. In 1970, Yuri Matiyasevich, building on a decades long line of work by Martin Davis, Hilary Putnam and Julia Robinson, showed that there is simply *no method* to solve such equations in general:

Theorem 10.10 — MRDP Theorem. Let $DIO : \{0, 1\}^* \rightarrow \{0, 1\}$ be the function that takes as input a string describing a 100-variable polynomial with integer coefficients $P(x_0, \dots, x_{99})$ and outputs 1 if and only if there exists $z_0, \dots, z_{99} \in \mathbb{N}$ s.t. $P(z_0, \dots, z_{99}) = 0$.

Then DIO is uncomputable.

As usual, we assume some standard way to express numbers and text as binary strings. The constant 100 is of course arbitrary; the problem is known to be uncomputable even for polynomials of degree four and at most 58 variables. In fact the number of variables can be reduced to nine, at the expense of the polynomial having a larger (but still constant) degree. See [Jones's paper](#) for more about this issue.

R

Remark 10.11 — Active code vs static data. The difficulty in finding a way to distinguish between “code” such as NAND-TM programs, and “static content” such as polynomials is just another manifestation of the phenomenon that *code* is the same as *data*. While a fool-proof solution for distinguishing between the two is inherently impossible, finding heuristics that do a reasonable job keeps many firewall and anti-virus manufacturers very busy (and finding ways to bypass these tools keeps many hackers busy as well).

95% of people cannot solve this!

$$\frac{\text{apple}}{\text{banana} + \text{orange}} + \frac{\text{banana}}{\text{apple} + \text{orange}} + \frac{\text{orange}}{\text{apple} + \text{banana}} = 4$$




Can you find positive whole values
for , , and ?

Figure 10.2: Diophantine equations such as finding a positive integer solution to the equation $a(a+b)(a+c) + b(b+a)(b+c) + c(c+a)(c+b) = 4(a+b)(a+c)(b+c)$ (depicted more compactly and whimsically above) can be surprisingly difficult. There are many equations for which we do not know if they have a solution, and there is no algorithm to solve them in general. The smallest solution for this equation has 80 digits! See this [Quora post](#) for more information, including the credits for this image.

³ This is a special case of what's known as “Fermat's Last Theorem” which states that $a^n + b^n = c^n$ has no solution in integers for $n > 2$. This was conjectured in 1637 by Pierre de Fermat but only proven by Andrew Wiles in 1991. The case $n = 11$ (along with all other so called “regular prime exponents”) was established by Kummer in 1850.

10.5 HARDNESS OF QUANTIFIED INTEGER STATEMENTS

We will not prove the MRDP Theorem (Theorem 10.10). However, as we mentioned, we will prove the uncomputability of *QIS* (i.e., Theorem 10.9), which is a special case of the MRDP Theorem. The reason is that a Diophantine equation is a special case of a quantified integer statement where the only quantifier is \exists . This means that deciding the truth of quantified integer statements is a potentially harder problem than solving Diophantine equations, and so it is potentially *easier* to prove that *QIS* is uncomputable.



If you find the last sentence confusing, it is worthwhile to reread it until you are sure you follow its logic. We are so accustomed to trying to find *solutions* for problems that it can sometimes be hard to follow the arguments for showing that problems are *uncomputable*.

Our proof of the uncomputability of *QIS* (i.e. Theorem 10.9) will, as usual, go by reduction from the Halting problem, but we will do so in two steps:

1. We will first use a reduction from the Halting problem to show that deciding the truth of *quantified mixed statements* is uncomputable. Quantified mixed statements involve both strings and integers. Since quantified mixed statements are a more general concept than quantified integer statements, it is *easier* to prove the uncomputability of deciding their truth.
2. We will then reduce the problem of quantified mixed statements to quantifier integer statements.

10.5.1 Step 1: Quantified mixed statements and computation histories

We define *quantified mixed statements* as statements involving not just integers and the usual arithmetic operators, but also *string variables* as well.

Definition 10.12 — Quantified mixed statements. A *quantified mixed statement* is a well-formed statement with no unbound variables involving integers, variables, the operators $>$, $<$, \times , $+$, $-$, $=$, the logical operations \neg (NOT), \wedge (AND), and \vee (OR), as well as quantifiers of the form $\exists_{x \in \mathbb{N}}$, $\exists_{a \in \{0,1\}^*}$, $\forall_{y \in \mathbb{N}}$, $\forall_{b \in \{0,1\}^*}$ where x, y, a, b are variable names. These also include the operator $|a|$ which returns the length of a string valued variable a , as well as the operator a_i where a is a string-valued variable and i is an integer valued ex-

pression which is true if i is smaller than the length of a and the i^{th} coordinate of a is 1, and is false otherwise.

For example, the true statement that for every string a there is a string b that corresponds to a in reverse order can be phrased as the following quantified mixed statement

$$\forall_{a \in \{0,1\}^*} \exists_{b \in \{0,1\}^*} (|a| = |b|) \wedge (\forall_{i \in \mathbb{N}} i < |a| \Rightarrow (a_i \Leftrightarrow b_{|a|-i})) . \quad (10.5)$$

Quantified mixed statements are more general than quantified integer statements, and so the following theorem is potentially easier to prove than [Theorem 10.9](#):

Theorem 10.13 — Uncomputability of quantified mixed statements. Let

$\text{QMS} : \{0,1\}^* \rightarrow \{0,1\}$ be the function that given a (string representation of) a quantified mixed statement outputs 1 if it is true and 0 if it is false. Then QMS is uncomputable.

Proof Idea:

The idea behind the proof is similar to that used in showing that one-dimensional cellular automata are Turing complete ([Theorem 7.7](#)) as well as showing that equivalence (or even “fullness”) of context free grammars is uncomputable ([Theorem 9.27](#)). We use the notion of a *configuration* of a NAND-TM program as in [Definition 7.8](#). Such a configuration can be thought of as a string α over some large-but-finite alphabet Σ describing its current state, including the values of all arrays, scalars, and the index variable i . It can be shown that if α is the configuration at a certain step of the execution and β is the configuration at the next step, then $\beta_j = \alpha_j$ for all j outside of $\{i-1, i, i+1\}$ where i is the value of i . In particular, every value β_j is simply a function of $\alpha_{j-1, j, j+1}$. Using these observations we can write a *quantified mixed statement* $\text{NEXT}(\alpha, \beta)$ that will be true if and only if β is the configuration encoding the next step after α . Since a program P halts on input x if and only if there is a sequence of configurations $\alpha^0, \dots, \alpha^{t-1}$ (known as a *computation history*) starting with the initial configuration with input x and ending in a halting configuration, we can define a quantified mixed statement to determine if there is such a statement by taking a universal quantifier over all strings H (for *history*) that encode a tuple $(\alpha^0, \alpha^1, \dots, \alpha^{t-1})$ and then checking that α^0 and α^{t-1} are valid starting and halting configurations, and that $\text{NEXT}(\alpha^j, \alpha^{j+1})$ is true for every $j \in \{0, \dots, t-2\}$.

★

Proof of Theorem 10.13. The proof is obtained by a reduction from the Halting problem. Specifically, we will use the notion of a *configuration* of a Turing Machines ([Definition 7.8](#)) that we have seen in the

context of proving that one dimensional cellular automata are Turing complete. We need the following facts about configurations:

- For every Turing Machine M , there is a finite alphabet Σ , and a *configuration* of M is a string $\alpha \in \Sigma^*$.
- A configuration α encodes all the state of the program at a particular iteration, including the array, scalar, and index variables.
- If α is a configuration, then $\beta = \text{NEXT}_P(\alpha)$ denotes the configuration of the computation after one more iteration. β is a string over Σ of length either $|\alpha|$ or $|\alpha| + 1$, and every coordinate of β is a function of just three coordinates in α . That is, for every $j \in \{0, \dots, |\beta| - 1\}$, $\beta_j = \text{MAP}_P(\alpha_{j-1}, \alpha_j, \alpha_{j+1})$ where $\text{MAP}_P : \Sigma^3 \rightarrow \Sigma$ is some function depending on P .
- There are simple conditions to check whether a string α is a valid starting configuration corresponding to an input x , as well as to check whether a string α is an halting configuration. In particular these conditions can be phrased as quantified mixed statements.
- A program M halts on input x if and only if there exists a sequence of configurations $H = (\alpha^0, \alpha^1, \dots, \alpha^{T-1})$ such that (i) α^0 is a valid starting configuration of M with input x , (ii) α^{T-1} is a valid halting configuration of P , and (iii) $\alpha^{i+1} = \text{NEXT}_P(\alpha^i)$ for every $i \in \{0, \dots, T-2\}$.

We can encode such a sequence H of configuration as a binary string. For concreteness, we let $\ell = \lceil \log(|\Sigma| + 1) \rceil$ and encode each symbol σ in $\Sigma \cup \{";"\}$ by a string in $\{0, 1\}^\ell$. We use ";" as a "separator" symbol, and so encode $H = (\alpha^0, \alpha^1, \dots, \alpha^{T-1})$ as the concatenation of the encodings of each configuration, using ";" to separate the encoding of α^i and α^{i+1} for every $i \in [T]$. In particular for every Turing Machine M , M halts on the input 0 if and only if the following statement φ_M is true

$$\exists_{H \in \{0,1\}^*} H \text{ encodes halting configuration sequence starting with input 0.} \quad (10.6)$$

If we can encode the statement φ_M as a mixed-integer statement then, since φ_M is true if and only if $\text{HALTONZERO}(M) = 1$, this would reduce the task of computing HALTONZERO to computing MIS , and hence imply (using [Theorem 8.10](#)) that MIS is uncomputable, completing the proof. Indeed, φ_M can be encoded as a mixed-integer statement for the following reasons:

1. Let $\alpha, \beta \in \{0, 1\}^*$ be two strings that encode configurations of M . We can define a quantified mixed predicate $\text{NEXT}(\alpha, \beta)$ that is true

if and only if $\beta = \text{NEXT}_M(\alpha)$ (i.e., β encodes the configuration obtained by proceeding from α in one computational step). Indeed $\text{NEXT}(\alpha, \beta)$ is true if **for every** $i \in \{0, \dots, |\beta|\}$ which is a multiple of ℓ , $\beta_{i, \dots, i+\ell-1} = \text{MAP}_M(\alpha_{i-\ell, \dots, i+2\ell-1})$ where $\text{MAP}_M : \{0, 1\}^{3\ell} \rightarrow \{0, 1\}^\ell$ is the finite function above (identifying elements of Σ with their encoding in $\{0, 1\}^\ell$). Since MAP_M is a finite function, we can express it using the logical operations *AND*, *OR*, *NOT* (for example by computing MAP_M with *NAND*'s).

2. Using the above we can now write the condition that **for every** substring of H that has the form $\alpha \text{ENC}(\cdot) \beta$ with $\alpha, \beta \in \{0, 1\}^\ell$ and $\text{ENC}(\cdot)$ being the encoding of the separator “;”, it holds that $\text{NEXT}(\alpha, \beta)$ is true.
3. Finally, if α^0 is a binary string encoding the initial configuration of M on input 0, checking that the first $|\alpha^0|$ bits of H equal α_0 can be expressed using *AND*, *OR*, and *NOT*'s. Similarly checking that the last configuration encoded by H corresponds to a state in which M will halt can also be expressed as a quantified statement.

Together the above yields a computable procedure that maps every Turing Machine M into a quantified mixed statement φ_M such that $\text{HALTONZERO}(M) = 1$ if and only if $\text{QMS}(\varphi_M) = 1$. This reduces computing HALTONZERO to computing QMS , and hence the uncomputability of HALTONZERO implies the uncomputability of QMS . ■



Remark 10.14 — Alternative proofs. There are several other ways to show that QMS is uncomputable. For example, we can express the condition that a 1-dimensional cellular automaton eventually writes a “1” to a given cell from a given initial configuration as a quantified mixed statement over a string encoding the history of all configurations. We can then use the fact that cellular automata can simulate Turing machines ([Theorem 7.7](#)) to reduce the halting problem to QMS . We can also use other well known uncomputable problems such as tiling or the [post correspondence problem](#). [Exercise 10.5](#) and [Exercise 10.6](#) explore two alternative proofs of [Theorem 10.13](#).

10.5.2 Step 2: Reducing mixed statements to integer statements

We now show how to prove [Theorem 10.9](#) using [Theorem 10.13](#). The idea is again a proof by reduction. We will show a transformation of every quantifier mixed statement φ into a quantified *integer* statement

ξ that does not use string-valued variables such that φ is true if and only if ξ is true.

To remove string-valued variables from a statement, we encode every string by a pair integer. We will show that we can encode a string $x \in \{0, 1\}^*$ by a pair of numbers $(X, n) \in \mathbb{N}$ s.t.

- $n = |x|$
- There is a quantified integer statement $COORD(X, i)$ that for every $i < n$, will be true if $x_i = 1$ and will be false otherwise.

This will mean that we can replace a “for all” quantifier over strings such as $\forall_{x \in \{0,1\}^*}$ with a pair of quantifiers over *integers* of the form $\forall_{X \in \mathbb{N}} \forall_{n \in \mathbb{N}}$ (and similarly replace an existential quantifier of the form $\exists_{x \in \{0,1\}^*}$ with a pair of quantifiers $\exists_{X \in \mathbb{N}} \exists_{n \in \mathbb{N}}$). We can then replace all calls to $|x|$ by n and all calls to x_i by $COORD(X, i)$. This means that if we are able to define $COORD$ via a quantified integer statement, then we obtain a proof of [Theorem 10.9](#), since we can use it to map every mixed quantified statement φ to an equivalent quantified integer statement ξ such that ξ is true if and only if φ is true, and hence $QMS(\varphi) = QIS(\xi)$. Such a procedure implies that the task of computing QMS reduces to the task of computing QIS , which means that the uncomputability of QMS implies the uncomputability of QIS .

The above shows that proof of [Theorem 10.9](#) all boils down to finding the right encoding of strings as integers, and the right way to implement $COORD$ as a quantified integer statement. To achieve this we use the following technical result :

Lemma 10.15 — Constructible prime sequence. There is a sequence of prime numbers $p_0 < p_1 < p_2 < \dots$ such that there is a quantified integer statement $PSEQ(p, i)$ that is true if and only if $p = p_i$.

Using [Lemma 10.15](#) we can encode a $x \in \{0, 1\}^*$ by the numbers (X, n) where $X = \prod_{x_i=1} p_i$ and $n = |x|$. We can then define the statement $COORD(X, i)$ as

$$COORD(X, i) = \exists_{p \in \mathbb{N}} PSEQ(p, i) \wedge DIVIDES(p, X) \quad (10.7)$$

where $DIVIDES(a, b)$, as before, is defined as $\exists_{c \in \mathbb{N}} a \times c = b$. Note that indeed if X, n encodes the string $x \in \{0, 1\}^*$, then for every $i < n$, $COORD(X, i) = x_i$, since p_i divides X if and only if $x_i = 1$.

Thus all that is left to conclude the proof of [Theorem 10.9](#) is to prove [Lemma 10.15](#), which we now proceed to do.

Proof. The sequence of prime numbers we consider is the following: We fix C to be a sufficiently large constant ($C = 2^{2^{34}}$ **will do**) and define p_i to be the smallest prime number that is in the interval $[(i + C)^3 + 1, (i + C + 1)^3 - 1]$. It is known that there exists such a prime

number for every $i \in \mathbb{N}$. Given this, the definition of $PSEQ(p, i)$ is simple:

$$(p > (i+C) \times (i+C) \times (i+C)) \wedge (p < (i+C+1) \times (i+C+1) \times (i+C+1)) \wedge (\forall_{p'} \neg \text{PRIME}(p') \vee (p' \leq i) \vee (p' \geq p)) , \quad (10.8)$$

We leave it to the reader to verify that $PSEQ(p, i)$ is true iff $p = p_i$. ■

To sum up we have shown that for every quantified mixed statement φ , we can compute a quantified integer statement ξ such that $QMS(\varphi) = 1$ if and only if $QIS(\xi) = 1$. Hence the uncomputability of QMS (Theorem 10.13) implies the uncomputability of QIS , completing the proof of Theorem 10.9, and so also the proof of Gödel's Incompleteness Theorem for quantified integer statements (Theorem 10.8).



Lecture Recap

- Uncomputable functions include also functions that seem to have nothing to do with NAND-TM programs or other computational models such as determining the satisfiability of diophantine equations.
- This also implies that for any sound proof system (and in particular every finite axiomatic system) S , there are interesting statements X (namely of the form " $F(x) = 0$ " for an uncomputable function F) such that S is not able to prove either X or its negation.

10.6 EXERCISES

Exercise 10.1 — Gödel's Theorem from uncomputability of QIS . Prove Theorem 10.8 using Theorem 10.9

Exercise 10.2 — Proof systems and uncomputability. Let $FINDPROOF : \{0, 1\}^* \rightarrow \{0, 1\}$ be the following function. On input a Turing machine V (which we think of as the verifying algorithm for a proof system) and a string $x \in \{0, 1\}^*$, $FINDPROOF(V, x) = 1$ if and only if there exists $w \in \{0, 1\}^*$ such that $V(x, w) = 1$.

1. Prove that $FINDPROOF$ is uncomputable.
2. Prove that there exists a Turing machine V such that V halts on every input x, v but the function $FINDPROOF_V$ defined as $FINDPROOF_V(x) = FINDPROOF(V, x)$ is uncomputable. See footnote for hint.⁴

⁴ Hint: think of x as saying "Turing Machine M halts on input u " and w being a proof that is the number of steps that it will take for this to happen. Can you find an always-halting V that will verify such statements?

Exercise 10.3 — Expression for floor. Let $FSQRT(n, m) = \forall_{j \in \mathbb{N}} ((j \times j) > m) \vee (j \leq n)$. Prove that $FSQRT(n, m)$ is true if and only if $n = \lfloor \sqrt{m} \rfloor$.

Exercise 10.4 — axiomatic proof systems. For every representation of logical statements as strings, we can define an axiomatic proof system to consist of a finite set of strings A and a finite set of rules I_0, \dots, I_{m-1} with $I_j : (\{0, 1\}^*)^{k_j} \rightarrow \{0, 1\}^*$ such that a proof (s_1, \dots, s_n) that s_n is true is valid if for every i , either $s_i \in A$ or is some $j \in [m]$ and are $i_1, \dots, i_{k_j} < i$ such that $s_i = I_j(s_{i_1}, \dots, s_{i_{k_j}})$. A system is *sound* if whenever there is no false s such that there is a proof that s is true. Prove that for every uncomputable function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ and every sound axiomatic proof system S (that is characterized by a finite number of axioms and inference rules), there is some input x for which the proof system S is not able to prove neither that $F(x) = 0$ nor that $F(x) \neq 0$.

Exercise 10.5 — Post Correspondence Problem. In the **Post Correspondence Problem** the input is a set $S = \{(\alpha^0, \beta^0), \dots, (\beta^{c-1}, \beta^{c-1})\}$ where each α^i and β^j is a string in $\{0, 1\}^*$. We say that $PCP(S) = 1$ if and only if there exists a list $(\alpha_0, \beta_0), \dots, (\alpha_{n-1}, \beta_{n-1})$ of pairs in S such that

$$\alpha_0 \alpha_1 \dots \alpha_{m-1} = \beta_0 \beta_1 \dots \beta_{m-1}. \quad (10.9)$$

(We can think of each pair $(\alpha, \beta) \in S$ as a “domino tile” and the question is whether we can stack a list of such tiles so that the top and the bottom yield the same string.) It can be shown that the PCP is uncomputable by a fairly straightforward though somewhat tedious proof (see for example the Wikipedia page for the Post Correspondence Problem or Section 5.2 in [Sip97]).

Use this fact to provide a direct proof that QMS is uncomputable by showing that there exists a computable map $R : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that $PCP(S) = QMS(R(S))$ for every string S encoding an instance of the post correspondence problem.

Exercise 10.6 — Uncomputability of puzzle. Let $PUZZLE : \{0, 1\}^* \rightarrow \{0, 1\}$ be the problem of determining, given a finite collection of types of “puzzle pieces”, whether it is possible to put them together in a rectangle, see Fig. 10.3. Formally, we think of such a collection as a finite set Σ (see Fig. 10.3). We model the criteria as to which pieces “fit together” by a pair of finite function $match_{\uparrow}, match_{\leftrightarrow} : \Sigma^2 \rightarrow \{0, 1\}$ such that a piece a fits above a piece b if and only if $match_{\uparrow}(a, b) = 1$ and a piece c fits to the left of a piece d if and only if $match_{\leftrightarrow}(c, d) = 1$. To model

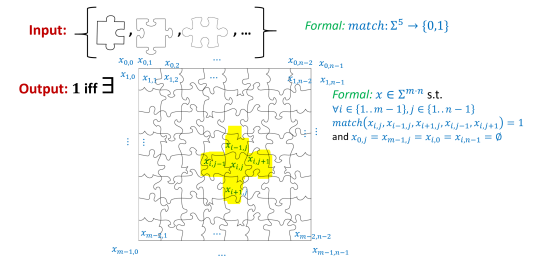


Figure 10.3: In the puzzle problem, the input can be thought of as a finite collection Σ of types of puzzle pieces and the goal is to find out whether or not find a way to arrange pieces from these types in a rectangle. Formally, we model the input as a pair of functions $match_{\leftrightarrow}, match_{\uparrow} : \Sigma^2 \rightarrow \{0, 1\}$ that such that $match_{\leftrightarrow}(left, right) = 1$ (respectively $match_{\uparrow}(up, down) = 1$) if the pair of pieces are compatible when placed in their respective positions. We assume Σ contains a special symbol \emptyset corresponding to having no piece, and an arrangement of puzzle pieces by an $(m-2) \times (n-2)$ rectangle is modeled by a string $x \in \Sigma^{m \times n}$ whose “outer coordinates” are \emptyset and such that for every $i \in [n-1], j \in [m-1]$, $match_{\uparrow}(x_{i,j}, x_{i+1,j}) = 1$ and $match_{\leftrightarrow}(x_{i,j}, x_{i,j+1}) = 1$.

the “straight edge” pieces that can be placed next to a “blank spot” we assume that Σ contains the symbol \emptyset and the matching functions are defined accordingly. A *square tiling* of Σ is an $m \times n$ long string $x \in \Sigma^{mn}$, such that for every $i \in \{1, \dots, m-2\}$ and $j \in \{1, \dots, n-2\}$, $\text{match}(x_{i,j}, x_{i-1,j}, x_{i+1,j}, x_{i,j-1}, x_{i,j+1}) = 1$ (i.e., every “internal pieve” fits in with the pieces adjacent to it). We also require all of the “outer pieces” (i.e., $x_{i,j}$ where $i \in \{0, m-1\}$ or $j \in \{0, n-1\}$) are “blank” or equal to \emptyset . The function *PUZZLE* takes as input a string describing the set Σ and the function *match* and outputs 1 if and only if there is some square tiling of Σ : some not all blank string $x \in \Sigma^{mn}$ satisfying the above condition.

1. Prove that *PUZZLE* is uncomputable.
2. Give a reduction from *PUZZLE* to *QMS*.

■

Exercise 10.7 — MRDP exercise. The MRDP theorem states that the problem of determining, given a k -variable polynomial p with integer coefficients, whether there exists integers x_0, \dots, x_{k-1} such that $p(x_0, \dots, x_{k-1}) = 0$ is uncomputable. Consider the following *quadratic integer equation problem*: the input is a list of polynomials p_0, \dots, p_{m-1} over k variables with integer coefficients, where each of the polynomials is of degree at most two (i.e., it is a *quadratic* function). The goal is to determine whether there exist integers x_0, \dots, x_{k-1} that solve the equations $p_0(x) = \dots = p_{m-1}(x) = 0$.

Use the MRDP Theorem to prove that this problem is uncomputable. That is, show that the function $\text{QUADINTEQ} : \{0, 1\}^* \rightarrow \{0, 1\}$ is uncomputable, where this function gets as input a string describing the polynomials p_0, \dots, p_{m-1} (each with integer coefficients and degree at most two), and outputs 1 if and only if there exists $x_0, \dots, x_{k-1} \in \mathbb{Z}$ such that for every $i \in [m]$, $p_i(x_0, \dots, x_{k-1}) = 0$. See footnote for hint⁵

⁵ You can replace the equation $y = x^4$ with the pair of equations $y = z^2$ and $z = x^2$. Also, you can replace the equation $w = x^6$ with the three equations $w = yu$, $y = x^4$ and $u = x^2$.

■

Exercise 10.8 — The Busy Beaver problem. In this question we define the NAND-TM variant of the *busy beaver function*.

1. We define the function $T : \{0, 1\}^* \rightarrow \mathbb{N}$ as follows: for every string $P \in \{0, 1\}^*$, if P represents a NAND-TM program such that when P is executed on the input 0 (i.e., the string of length 1 that is simply 0), a total of M lines are executed before the program halts, then $T(P) = M$. Otherwise (if P does not represent a NAND-TM program, or it is a program that does not halt on 0), $T(P) = 0$. Prove that T is uncomputable.

2. Let $TOWER(n)$ denote the number $\underbrace{2^{2^2 \cdots 2}}_{n \text{ times}}$ (that is, a “tower of powers of two” of height n). To get a sense of how fast this function grows, $TOWER(1) = 2$, $TOWER(2) = 2^2 = 4$, $TOWER(3) = 2^{2^2} = 16$, $TOWER(4) = 2^{16} = 65536$ and $TOWER(5) = 2^{65536}$ which is about 10^{20000} . $TOWER(6)$ is already a number that is too big to write even in scientific notation. Define $NBB : \mathbb{N} \rightarrow \mathbb{N}$ (for “NAND-TM Busy Beaver”) to be the function $NBB(n) = \max_{P \in \{0,1\}^n} T(P)$ where $T : \mathbb{N} \rightarrow \mathbb{N}$ is the function defined in Item 1. Prove that NBB grows *faster* than $TOWER$, in the sense that $TOWER(n) = o(NBB(n))$ (i.e., for every $\epsilon > 0$, there exists n_0 such that for every $n > n_0$, $TOWER(n) < \epsilon \cdot NBB(n)$).⁶

⁶ You will not need to use very specific properties of the $TOWER$ function in this exercise. For example, $NBB(n)$ also grows faster than the [Ackerman function](#). You might find [Aaronson’s blog post](#) on the same topic to be quite interesting, and relevant to this book at large. If you like it then you might also enjoy [this piece by Terence Tao](#).

10.7 BIBLIOGRAPHICAL NOTES

As mentioned before, Gödel, Escher, Bach [[Hof99](#)] is a highly recommended book covering Gödel’s Theorem. A classic popular science book about Fermat’s Last Theorem is [[Sin97](#)].

Cantor’s are used for both Turing’s and Gödel’s theorems. In a twist of fate, using techniques originating from the works Gödel and Turing, Paul Cohen showed in 1963 that Cantor’s *Continuum Hypothesis* is independent of the axioms of set theory, which means that neither it nor its negation is provable from these axioms and hence in some sense can be considered as “neither true nor false” (see [[Coh08](#)]). The *Continuum Hypothesis* is the conjecture that for every subset S of \mathbb{R} , either there is a one-to-one and onto map between S and \mathbb{N} or there is a one-to-one and onto map between S and \mathbb{R} . It was conjectured by Cantor and listed by Hilbert in 1900 as one of the most important problems in mathematics. See also the non-conventional survey of Shelah [[She03](#)]. See [here](#) for recent progress on a related question.

Thanks to Alex Lombardi for pointing out an embarrassing mistake in the description of Fermat’s Last Theorem. (I said that it was open for exponent 11 before Wiles’ work.)



EFFICIENT ALGORITHMS

