Is every theorem provable?

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Is every theorem provable?

### 

* 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.

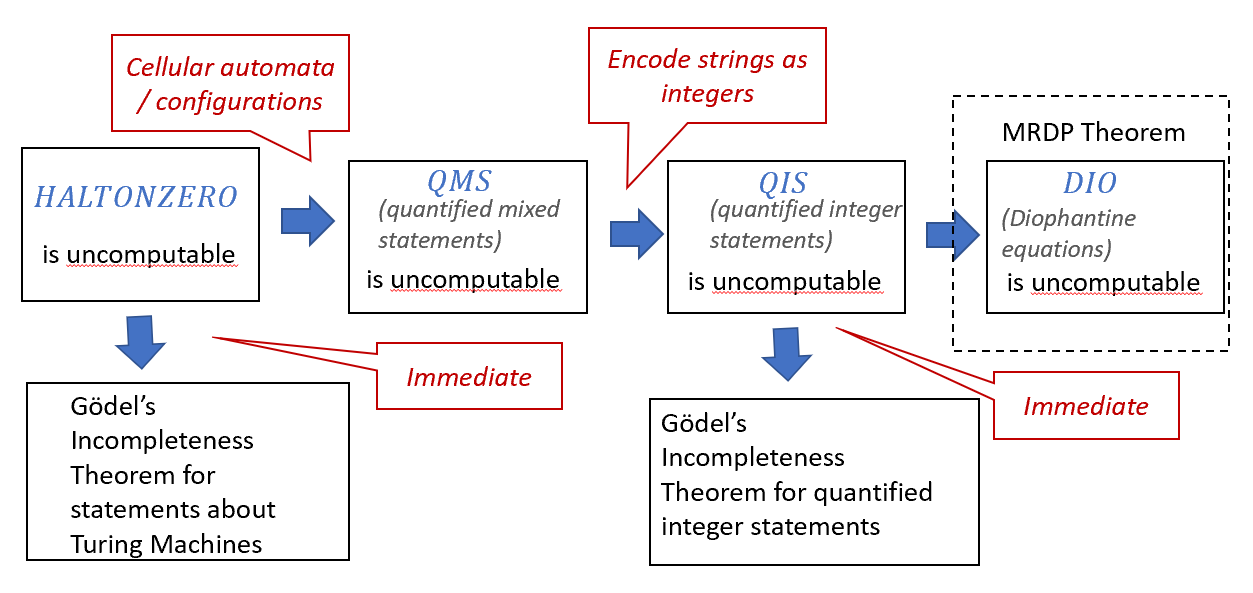
*“Take any definite unsolved problem, such as … the existence of an infinite number of prime numbers of the form . However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number 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 chapcomputable, 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.

The marquee result of this chapter is Gödel’s Incompleteness Theorem, which states that for every proof system, there are some statements about arithmetic that are true but *unprovable* in this system. But more than that we will see a deep connection between *uncomputability* and *unprovability*. For example, the uncomputability of the Halting problem immediately gives rise to the existence of unprovable statements about Turing machines. To even state Gödel’s Incompleteness Theorem we will need to formally define the notion of a “proof system”. We give a very general definition, that encompasses all types of “axioms + inference rules” systems used in logic and math. We will then build up the machinery to encode computation using arithmetic that will enable us to prove Gödel’s Theorem.



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 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 alternativeproofs.

## 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 ) 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 the works of mathematicians such as Cauchy, Weierstrass, and Riemann, who eventually placed analysis on firmer foundations, giving rise to the ’s and ’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:

### 

For every sound proof system for sufficiently rich mathematical statements, there is a mathematical statement that is *true* but is not *provable* in .

### Defining “Proof Systems”

Before proving godethminformal, 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](https://en.wikipedia.org/wiki/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* . 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]](#footnote-29)

*“The number ,,,,,,,,,,, ,,,,,,,,,, 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]](#footnote-30)

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 is another piece of text that certifies the truth of the statement asserted in . The conditions for a valid proof system are:

1. *(Effectiveness)* Given a statement and a proof , there is an algorithm to verify whether or not is a valid proof for . (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 for then 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 a system where the “proof” for a statement is “trust me: it’s true”.

We formally define proof systems as an algorithm where holds if the string is a valid proof for the statement . Even if is true, the string 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 is a valid proof for then must be true.

Let be some set (which we consider the “true” statements). A *proof system* for is an algorithm that satisfies:

1. *(Effectiveness)* For every , halts with an output of either or .
2. *(Soundness)* For every and , .

A true statement is *unprovable* (with respect to ) if for every , . We say that is *complete* if there does not exist a true statement that is unprovable with respect to .

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

## Gödel’s Incompleteness Theorem: Computational variant

Our first formalization of godethminformal involves statements about Turing machines. We let be the set of strings that have the form “Turing machine does not halt on the zero input”.

There does not exist a complete proof system for .

### 

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

Assume for the sake of contradiction that there was such a proof system . We will use to build an algorithm that computes , hence contradicting haltonzero-thm. Our algorithm will work as follows:

INPUT: Turing machine $M$  
OUTPUT: $1$ $M$ -if halts on -input $0$; $0$ otherwise.  
  
for{$n=1,2,3,\ldots$}  
 for{$w\in \{0,1\}^n$}  
 if{$V($"$M$ does not halt on $0$"$,w)=1$}  
 return $0$  
 endif  
 Simulate $M$ -for $n$ steps on $0$.  
 if{$M$ halts}  
 return $1$  
 endif  
 endfor  
endfor

If halts on zero within steps, then by the soundness of the proof system, there will not exist a proof for “ does not halt on ” on so we will never return . By the time time we get to in the loop, we will simulate for steps and so return . On the hand, if does not halt on , then we will never return . Because the proof system is complete, there exists that proves this fact, and so when Algorithm reaches we will eventually find this and output . Hence under the assumption that the proof system is complete and sound, solves the function, yielding a contradiction.

One can extract from the proof of godethmtakeone a procedure that for every proof system , yields a true statement that cannot be proven in . But Gödel’s proof gave a very explicit description of such a statement which is closely related to the [“Liar’s paradox”](https://en.wikipedia.org/wiki/Liar_paradox). That is, Gödel’s statement was designed to be true if and only if . In other words, it satisfied the following property

One can see that if is true, then it does not have a proof, but if it is false then (assuming the proof system is sound) then it cannot have a proof, and hence must be both true and unprovable. One might wonder how is it possible to come up with an that satisfies a condition such as godeleq where the same string appears on both the right-hand side and the left-hand side of the equation. The idea is that the proof of godethmtakeone yields a way to transform every statement into a statement that is true if and only if does not have a proof in . Thus needs to be a *fixed point* of : a sentence such that . It turns out that [we can always find](https://en.wikipedia.org/wiki/Kleene%27s_recursion_theorem) such a fixed point of . We’ve already seen this phenomenon in the calculus, where the combinator maps every into a fixed point of . 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 . “The following sentence repeated twice, the second time in quotes, is not provable in the formal system .”

In the argument above we actually showed that is *true*, under the assumption that is sound. Since is true and does not have a proof in , this means that we cannot carry the above argument in the system , which means that 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 cannot prove its own consistency. That is, if we formalize the statement that is true if and only if is consistent (i.e., cannot prove both a statement and the statement’s negation), then cannot be proven in .

## Quantified integer statements

There is something “unsatisfying” about godethmtakeone. 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 and such that ”, or “there are positive integers such that ” 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](https://goo.gl/qsU9zy).)

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

### 

A *quantified integer statement* is a well-formed statement with no unbound variables involving integers, variables, the operators , the logical operations (NOT), (AND), and (OR), as well as quantifiers of the form and where are variable names.

We often care deeply about determining the truth of quantified integer statements. For example, the statement that [Fermat’s Last Theorem](https://goo.gl/fvkuqj) is true for can be phrased as the quantified integer statement

The [twin prime conjecture](https://goo.gl/GRiVz3), that states that there is an infinite number of numbers such that both and are primes can be phrased as the quantified integer statement

where we replace an instance of with the statement .

The claim (mentioned in Hilbert’s quote above) that are infinitely many primes of the form can be phrased as follows:

where is the statement . In English, this corresponds to the claim that for every there is some such that all of ’s prime factors are equal to .

To make our statements more readable, we often use syntactic sugar and so write as shorthand for , and so on. Similarly, the “implication operator” is “syntactic sugar” or shorthand for , and the “if and only if operator” is shorthand for ). We will also allow ourselves the use of “macros”: plugging in one quantified integer statement in another, as we did with and 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:

Let a computable purported verification procedure for quantified integer statements. Then either:

* *is not sound:* There exists a false statement and a string such that .

*or*

* *is not complete:* There exists a true statement such that for every , .

godelthmqis is a direct corollary of the following result, just as godethmtakeone was a direct corollary of the uncomputability of :

### 

Let be the function that given a (string representation of) a quantified integer statement outputs if it is true and if it is false. Then 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 .

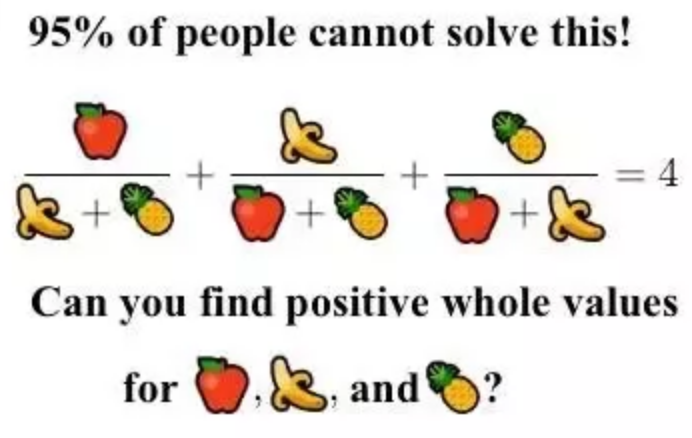
Please stop here and make sure you understand why the uncomputability of (i.e., QIS-thm) means that there is no sound and complete proof system for proving quantified integer statements (i.e., godelthmqis). This follows in the same way that godethmtakeone followed from the uncomputability of , but working out the details is a great exercise (see godelfromqisex)

In the rest of this chapter, we will show the proof of godelthmqis, following the outline illustrated in godelstructurefig.

## 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](https://en.wikipedia.org/wiki/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 and ) 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](https://en.wikipedia.org/wiki/The_Nine_Chapters_on_the_Mathematical_Art) from the first century mentions such approaches. Solving polynomial equations is by no means restricted only to ancient history or to students’ homework. The [gradient descent](https://en.wikipedia.org/wiki/Gradient_descent) method is the workhorse powering many of the machine learning tools that have revolutionized Computer Science over the last several years.



Diophantine equations such as finding a positive integer solution to the equation (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 digits! See this [Quora post](https://www.quora.com/How-do-you-find-the-positive-integer-solutions-to-frac-x-y+z-+-frac-y-z+x-+-frac-z-x+y-4) for more information, including the credits for this image.

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 has no solution in integers.[[3]](#footnote-56) 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:

Let be the function that takes as input a string describing a -variable polynomial with integer coefficients and outputs if and only if there exists s.t. .

Then is uncomputable.

As usual, we assume some standard way to express numbers and text as binary strings. The constant 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](https://www.jstor.org/stable/2273588) for more about this issue.

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).

## Hardness of quantified integer statements

We will not prove the MRDP Theorem (MRDP-thm). However, as we mentioned, we will prove the uncomputability of (i.e., QIS-thm), 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 . 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 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 (i.e. QIS-thm) 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 quantified integer statements.

### 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.

A *quantified mixed statement* is a well-formed statement with no unbound variables involving integers, variables, the operators , the logical operations (NOT), (AND), and (OR), as well as quantifiers of the form , , , where are variable names. These also include the operator which returns the length of a string valued variable , as well as the operator where is a string-valued variable and is an integer valued expression which is true if is smaller than the length of and the coordinate of is , and is false otherwise.

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

Quantified mixed statements are more general than quantified integer statements, and so the following theorem is potentially easier to prove than QIS-thm:

### 

Let be the function that given a (string representation of) a quantified mixed statement outputs if it is true and if it is false. Then is uncomputable.

### 

The idea behind the proof is similar to that used in showing that one-dimensional cellular automata are Turing complete (onedimcathm) as well as showing that equivalence (or even “fullness”) of context free grammars is uncomputable (fullnesscfgdef). We use the notion of a *configuration* of a NAND-TM program as in configtmdef. Such a configuration can be thought of as a string over some large-but-finite alphabet describing its current state, including the values of all arrays, scalars, and the index variable i. It can be shown that if is the configuration at a certain step of the execution and is the configuration at the next step, then for all outside of where is the value of i. In particular, every value is simply a function of . Using these observations we can write a *quantified mixed statement* that will be true if and only if is the configuration encoding the next step after . Since a program halts on input if and only if there is a sequence of configurations (known as a *computation history*) starting with the initial configuration with input 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 (for *history*) that encode a tuple and then checking that and are valid starting and halting configurations, and that is true for every .

The proof is obtained by a reduction from the Halting problem. Specifically, we will use the notion of a *configuration* of a Turing machines (configtmdef) 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 , there is a finite alphabet , and a *configuration* of is a string .
* A configuration encodes all the state of the program at a particular iteration, including the array, scalar, and index variables.
* If is a configuration, then denotes the configuration of the computation after one more iteration. is a string over of length either or , and every coordinate of is a function of just three coordinates in . That is, for every , where is some function depending on .
* There are simple conditions to check whether a string is a valid starting configuration corresponding to an input , as well as to check whether a string is a halting configuration. In particular these conditions can be phrased as quantified mixed statements.
* A program halts on input if and only if there exists a sequence of configurations such that **(i)** is a valid starting configuration of with input , **(ii)** is a valid halting configuration of , and **(iii)** for every .

We can encode such a sequence of configuration as a binary string. For concreteness, we let and encode each symbol in $\Sigma \cup \{ ";" \}$ by a string in . We use “” as a “separator” symbol, and so encode as the concatenation of the encodings of each configuration, using “” to separate the encoding of and for every . In particular for every Turing machine , halts on the input if and only if the following statement is true

If we can encode the statement as a quantified mixed statement then, since is true if and only if , this would reduce the task of computing to computing , and hence imply (using haltonzero-thm ) that is uncomputable, completing the proof. Indeed, can be encoded as a quantified mixed statement for the following reasons:

1. Let be two strings that encode configurations of . We can define a quantified mixed predicate that is true if and only if (i.e., encodes the configuration obtained by proceeding from in one computational step). Indeed is true if **for every** which is a multiple of , where is the finite function above (identifying elements of with their encoding in ). Since is a finite function, we can express it using the logical operations ,, (for example by computing with ’s).
2. Using the above we can now write the condition that **for every** substring of that has the form with and being the encoding of the separator “”, it holds that is true.
3. Finally, if is a binary string encoding the initial configuration of on input , checking that the first bits of equal can be expressed using ,, and ’s. Similarly checking that the last configuration encoded by corresponds to a state in which will halt can also be expressed as a quantified statement.

Together the above yields a computable procedure that maps every Turing machine into a quantified mixed statement such that if and only if . This reduces computing to computing , and hence the uncomputability of implies the uncomputability of .

There are several other ways to show that is uncomputable. For example, we can express the condition that a 1-dimensional cellular automaton eventually writes a “” 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 automatons can simulate Turing machines (onedimcathm) to reduce the halting problem to . We can also use other well known uncomputable problems such as tiling or the [post correspondence problem](https://en.wikipedia.org/wiki/Post_correspondence_problem). postcorrespondenceproblemex and puzzleex explore two alternative proofs of QMS-thm.

### Step 2: Reducing mixed statements to integer statements

We now show how to prove QIS-thm using QMS-thm. The idea is again a proof by reduction. We will show a transformation of every quantified mixed statement into a quantified *integer* statement that does not use string-valued variables such that is true if and only if 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 by a pair of numbers s.t.

* There is a quantified integer statement that for every , will be true if and will be false otherwise.

This will mean that we can replace a “for all” quantifier over strings such as with a pair of quantifiers over *integers* of the form (and similarly replace an existential quantifier of the form with a pair of quantifiers ) . We can then replace all calls to by and all calls to by . This means that if we are able to define via a quantified integer statement, then we obtain a proof of QIS-thm, since we can use it to map every mixed quantified statement to an equivalent quantified integer statement such that is true if and only if is true, and hence . Such a procedure implies that the task of computing reduces to the task of computing , which means that the uncomputability of implies the uncomputability of .

The above shows that proof of QIS-thm all boils down to finding the right encoding of strings as integers, and the right way to implement as a quantified integer statement. To achieve this we use the following technical result :

### 

There is a sequence of prime numbers such that there is a quantified integer statement that is true if and only if .

Using primeseq we can encode a by the numbers where and . We can then define the statement as

where , as before, is defined as . Note that indeed if encodes the string , then for every , , since divides if and only if .

Thus all that is left to conclude the proof of QIS-thm is to prove primeseq, which we now proceed to do.

The sequence of prime numbers we consider is the following: We fix to be a sufficiently large constant ( [will do](https://arxiv.org/pdf/1401.4233.pdf)) and define to be the smallest prime number that is in the interval . It is known that there exists such a prime number for every . Given this, the definition of is simple:

We leave it to the reader to verify that is true iff .

To sum up we have shown that for every quantified mixed statement , we can compute a quantified integer statement such that if and only if . Hence the uncomputability of (QMS-thm) implies the uncomputability of , completing the proof of QIS-thm, and so also the proof of Gödel’s Incompleteness Theorem for quantified integer statements (godelthmqis).

### 

* 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) , there are interesting statements (namely of the form “” for an uncomputable function ) such that is not able to prove either or its negation.

## Exercises

Prove godelthmqis using QIS-thm.

Let be the following function. On input a Turing machine (which we think of as the verifying algorithm for a proof system) and a string , if and only if there exists such that .

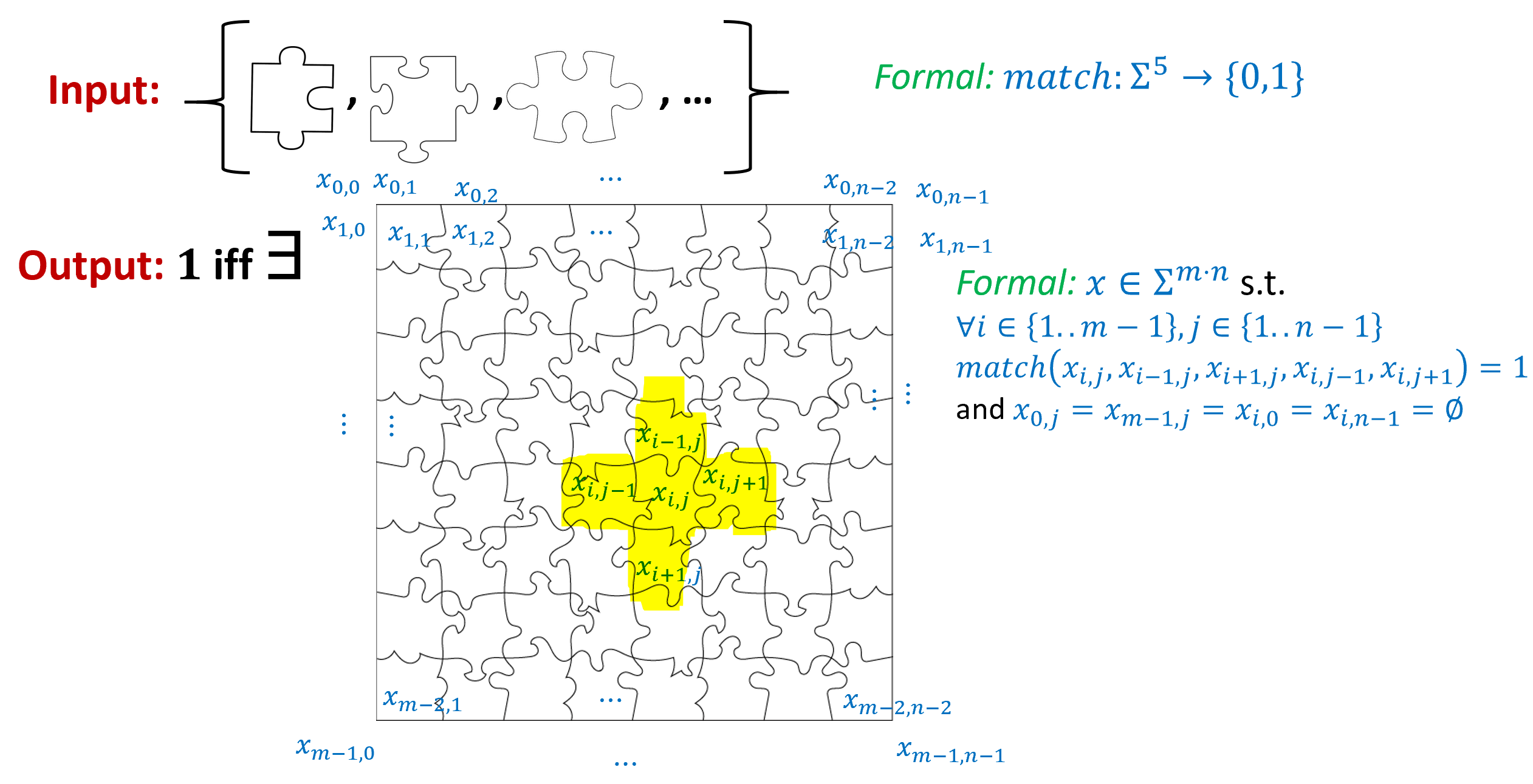
1. Prove that is uncomputable.
2. Prove that there exists a Turing machine such that *halts on every input*  but the function defined as is uncomputable. See footnote for hint.[[4]](#footnote-73)

### 

Let . Prove that is true if and only if .

### 

For every representation of logical statements as strings, we can define an axiomatic proof system to consist of a finite set of strings and a finite set of rules with such that a proof that is true is valid if for every , either or is some and are such that . A system is *sound* if whenever there is no false such that there is a proof that is true. Prove that for every uncomputable function and every sound axiomatic proof system (that is characterized by a finite number of axioms and inference rules), there is some input for which the proof system is not able to prove neither that nor that .



In the *puzzle problem*, the input can be thought of as a finite collection 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 that such that (respectively ) if the pair of pieces are compatible when placed in their respective positions. We assume contains a special symbol corresponding to having no piece, and an arrangement of puzzle pieces by an rectangle is modeled by a string whose ``outer coordinates’’ are and such that for every , and .

In the [Post Correspondence Problem](https://en.wikipedia.org/wiki/Post_correspondence_problem) the input is a set where each and is a string in . We say that if and only if there exists a list of pairs in such that

(We can think of each pair 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 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 [@SipserBook]).

Use this fact to provide a direct proof that is uncomputable by showing that there exists a computable map such that for every string encoding an instance of the post correspondence problem.

Let 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 puzzleprobfig. Formally, we think of such a collection as a finite set (see puzzleprobfig). We model the criteria as to which pieces “fit together” by a pair of finite function such that a piece fits above a piece if and only if and a piece fits to the left of a piece if and only if . To model the “straight edge” pieces that can be placed next to a “blank spot” we assume that contains the symbol and the matching functions are defined accordingly. A *square tiling* of is an long string , such that for every and , (i.e., every “internal pieve” fits in with the pieces adjacent to it). We also require all of the “outer pieces” (i.e., where of ) are “blank” or equal to . The function takes as input a string describing the set and the function and outputs if and only if there is some square tiling of : some not all blank string satisfying the above condition.

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

The MRDP theorem states that the problem of determining, given a -variable polynomial with integer coefficients, whether there exists integers such that is uncomputable. Consider the following *quadratic integer equation problem*: the input is a list of polynomials over 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 that solve the equations .

Use the MRDP Theorem to prove that this problem is uncomputable. That is, show that the function is uncomputable, where this function gets as input a string describing the polynomials (each with integer coefficients and degree at most two), and outputs if and only if there exists such that for every , . See footnote for hint[[5]](#footnote-81)

In this question we define the NAND-TM variant of the [busy beaver function](https://www.scottaaronson.com/writings/bignumbers.html).

1. We define the function as follows: for every string , if represents a NAND-TM program such that when is executed on the input (i.e., the string of length 1 that is simply ), a total of lines are executed before the program halts, then . Otherwise (if does not represent a NAND-TM program, or it is a program that does not halt on ), . Prove that is uncomputable.
2. Let denote the number (that is, a “tower of powers of two” of height ). To get a sense of how fast this function grows, , , , and which is about . is already a number that is too big to write even in scientific notation. Define (for “NAND-TM Busy Beaver”) to be the function where is the function defined in Item 1. Prove that grows *faster* than , in the sense that (i.e., for every , there exists such that for every , .).[[6]](#footnote-84)

## Bibliographical notes

As mentioned before, Gödel, Escher, Bach [@hofstadter1999] is a highly recommended book covering Gödel’s Theorem. A classic popular science book about Fermat’s Last Theorem is [@singh1997fermat].

Cantor’s are used for both Turing and Gödel’s theorems. In a twist of fate, using techniques originating from the works of 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 [@cohen2008set]). The [Continuum Hypothesis](https://goo.gl/9ieBVq) is the conjecture that for every subset of , either there is a one-to-one and onto map between and or there is a one-to-one and onto map between and . 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 [@shelah2003logical]. See [here](https://gowers.wordpress.com/2017/09/19/two-infinities-that-are-surprisingly-equal/) 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.)

1. This happens to be a *false* statement. [↑](#footnote-ref-29)
2. It is [unknown](https://goo.gl/Lx8HYv) whether this statement is true or false. [↑](#footnote-ref-30)
3. This is a special case of what’s known as “Fermat’s Last Theorem” which states that has no solution in integers for . This was conjectured in 1637 by Pierre de Fermat but only proven by Andrew Wiles in 1991. The case (along with all other so called “regular prime exponents”) was established by Kummer in 1850. [↑](#footnote-ref-56)
4. *Hint:* think of as saying “Turing machine halts on input ” and being a proof that is the number of steps that it will take for this to happen. Can you find an always-halting that will verify such statements? [↑](#footnote-ref-73)
5. You can replace the equation with the pair of equations and . Also, you can replace the equation with the three equations , and . [↑](#footnote-ref-81)
6. You will not need to use very specific properties of the function in this exercise. For example, also grows faster than the [Ackerman function](https://en.wikipedia.org/wiki/Ackermann_function). You might find [Aaronson’s blog post](https://www.scottaaronson.com/blog/?p=3445) 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](https://terrytao.wordpress.com/2010/10/10/the-cosmic-distance-ladder-ver-4-1/). [↑](#footnote-ref-84)