Learning Objectives:

- See more examples of uncomputable functions that are not as tied to computation.
- See 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++ 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.

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 $1 - 1 + 1 - 1 + 1 + \ldots$) 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 for sufficiently rich mathematical statements, there is a mathematical statement that is *true* but is not *provable*.

Before proving Theorem 10.2, we need to specify what does it mean to be "provable" (and even formally define the notion of a "mathematical statement"). Thus we need to define the notion of a *proof system*. 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?

Our idea will be to use an extremely general notion of proof, not even restricting ourselves to ones that have the form of axioms and inference. A *proof* will be simply a piece of text- a finite string- that satisfies:

1. *(effectiveness)* Given a statement x and a proof w (both of which can be encoded as strings) we can verify that w is a valid proof for x. (For example, by going line by line and checking that each line does indeed follow 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.

Those seem like rather minimal requirements that one would want from every 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 it 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 would simply be "trust me: it's true".

A mathematical statement will also simply be a string. Mathematical statements states a fact about some mathematical object. For example, the following is a mathematical statement:

> *"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".*

(This happens to be a *false* statement; can you see why?)

Mathematical statements don't have to be about numbers. They can talk about any other mathematical object including sets, strings, functions, graphs and yes, even *programs*. Thus, another example of a mathematical statement is the following:

> 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)
> ```

(We actually don't know if this statement is true or false.)

We start by considering statements of the second type. Our first formalization of Theorem 10.2 will be the following

Theorem 10.2 — Gödel's Incompleteness Theorem: computational variant. Let $V : \{0,1\}^* \rightarrow \{0,1\}$ a computable purported verification procedure for mathematical statements of the form "Program P halts on the zero input" and "Program P does not halt on the zero input". 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 $w \in \{0,1\}^*$, $V(x,w) = 0$.

Proof Idea: If we had such a complete and sound proof system then we could solve the $HALTONZERO$ problem. On input a program P, we would search all purported proofs w and halt as soon as we find a proof of either "P halts on zero" or "P 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 10.2. 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 8.4. Our algorithm A will will work as follows:

Algorithm A:

- **Input:** NAND++ program P
- **Goal:** Determine if P halts on the input 0.
- **Assumption:** We have access to a proof system V such that for every statement x of the form "Program Q halts on 0" or "Program Q does not halt on 0", there exists some string $w \in \{0,1\}^*$ such that $V(x,w) = 1$ if and only if x is true.

Operation:

```
• For $n = 0, 1, 2, ...$:
  – For $w \in \{0,1\}^n$:
    * If $V(\text{"}P \text{ halts on } 0\text{"}, w) = 1$ output 1
    * If $V(\text{"}P \text{ does not halt on } 0\text{"}, w) = 1$ output 0
```

If P 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 P 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 **The Gödel statement (optional)** One can extract from the proof of Theorem 10.2 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

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

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 Eq. (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 **??** 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.

10.2 QUANTIFIED INTEGER STATEMENTS

There is something "unsatisfying" about Theorem 10.2. 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 integers x and y such that $x^2 - 2 = y^7$", or "there are integers x, y, z such that $x^2 + y^6 = z^{11}$" that only talk about *natural numbers*.[1] 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."

[1] I do not know if these statements are actually true or false, see here.

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

Definition 10.3 — 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

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

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

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

where we replace an instance of $PRIME(q)$ 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 are infinitely many primes of the form $p = 2^n + 1$ can be phrased as follows:

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

where $DIVIDES(a, b)$ 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.

R **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" $a \Leftrightarrow$ 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 10.4 — 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 10.4 is a direct corollary of the following result, just as Theorem 10.2 was a direct corollary of the uncomputability of $HALTONZERO$:

Theorem 10.5 — Uncomputability of quantified integer statements. Let $QIS : \{0,1\}^* \to \{0,1\}$ 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. [2] Then QIS is uncomputable.

P Please stop here and make sure you understand why the uncomputability of QIS (i.e., Theorem 10.5) means that there is no sound and complete proof system for proving quantified integer statements (i.e., Theorem 10.4). This follows in the same way that Theorem 10.2 followed from the uncomputability of $HALTONZERO$, but working out the details is a great exercise (see Exercise 10.1)

[2] Since a quantified integer statement is simply a sequence of symbols, we can easily represent it as a string. 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 $\exists_{x\in\mathbb{N}} x = 1$.

In the rest of this chapter, we will show the proof of **??**.

10.3 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

[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.

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 10.6 — 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 $P(x_0, \ldots, x_{99})$ and outputs 1 if and only if there exists $z_0, \ldots, z_{99} \in \mathbb{N}$ s.t. $P(z_0, \ldots, z_{99}) = 0$.

Then DIO is uncomputable. [4]

[4] 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 **Active code vs static data** The difficulty in finding a way to distinguish between "code" such as NAND++ 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).

10.4 HARDNESS OF QUANTIFIED INTEGER STATEMENTS

We will not prove the MRDP Theorem (Theorem 10.6). However, as we mentioned, we will prove the uncomputability of QIS (i.e., Theorem 10.5), 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.

P 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 10.5) 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. Unquantified 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.

10.4.1 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 10.7 — 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 expression 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 10.5:

Theorem 10.8 — Uncomputability of quantified mixed statements. Let $QMS : \{0,1\}^* \rightarrow \{0,1\}$ 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 7.11) as well as showing that equivalence (or even "fullness") of context free grammars is uncomputable (Theorem 9.20). We use the notion of a *configuration* of a NAND++ program as in Definition 7.12. 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* $NEXT(\alpha, \beta)$ 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 $\alpha^0, \ldots, \alpha^{t-1}$ (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 $(\alpha^0, \alpha^1, \ldots, \alpha^{t-1})$ and then checking that α^0 and α^{t-1} are valid starting and halting configurations, and that $NEXT(\alpha^j, \alpha^{j+1})$ is true for every $j \in \{0, \ldots, t-2\}$. ⋆

Proof of Theorem 10.8. The proof will be obtained by a reduction from the Halting problem. Specifically, we will use the notion of a *configuration* of a NAND++ program (Definition 7.12) 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 (well formed[5]) NAND++ program P, there is a finite alphabet Σ, and a *configuration* of P 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 $\beta = NEXT_P(\alpha)$ 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 $j \in \{0, \ldots, |\beta|-1\}$,

[5] We can always transform a NAND++ program into an equivalent one that is well formed (see Lemma 6.9), and hence can assume this property without loss of generality.

$\beta_j = MAP_P(\alpha_{j-1}, \alpha_j, \alpha_{j+1})$ where $MAP_P : \Sigma^3 \to \Sigma$ is some function depending on P.[6]

- 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 P halts on input x if and only if there exists a sequence of configurations $H = (\alpha^0, \alpha^1, \ldots, \alpha^{T-1})$ such that **(i)** α^0 is a valid starting configuration of P with input x, **(ii)** α^{T-1} is a valid halting configuration of P, and **(iii)** $\alpha^{i+1} = NEXT_P(\alpha^i)$ for every $i \in \{0, \ldots, T-2\}$.

[6] The alphabet Σ contains a special "default" element, which we can denote by $\varnothing$, such that if $j - 1 < 0$ or j or $j + 1$ are at least $|\alpha|$, we use $\varnothing$ as input instead of α_{j-1} or α_{j+1} respectively. We extend the length of β to be one longer than α if and only if $N_P(\alpha_{|\alpha|-1}, \varnothing, \varnothing) \neq \varnothing$.

Let U be a universal NAND++ program. Such a program exists by Theorem 8.1. We define $HALT_U$ as the function such that $HALT_U(w) = 1$ if and only if U halts on the input w. We claim that the function $HALT_U$ is uncomputable. Indeed, for every NAND++ program P (which we identify with its representation as a string) and input $x \in \{0,1\}^*$ to P, $HALT(P, x) = HALT_U(\langle P, x \rangle)$ where $\langle P, x \rangle$ is some encoding of the pair (P, x) as a string. Hence if we could compute $HALT_U$ then we could compute $HALT$, contradicting Theorem 8.3.

Let Σ be the alphabet needed to encode configurations of U, and let $\ell = \lceil \log(|\Sigma| + 1) \rceil$. Then we can encode any symbol in $\Sigma \cup \{;\}$ (where ";" is some separator symbol we'll use) as a string in $\{0,1\}^\ell$, and so in particular can encode a sequence $\alpha^0; \alpha^1; \cdots; \alpha^T$ of configurations of U as a single binary string which we'll also name as H. Given any input $w \in \{0,1\}^*$, we will create a mixed integer statement φ_w that will have the following form:

$$\varphi_w = \exists_{H \in \{0,1\}^*} H \text{ encodes a valid sequence of configurations of a halting computation of } U \text{ on } w \tag{10.6}$$

The reasons we can encode this condition as an MIS are the following:

1. The conditions for checking that the initial configuration is valid are simple, and we can extract the first configuration from H by first looking at an index i which is a multiple of ℓ such that $H_i \cdots H_{i+\ell-1}$ encodes the separator symbol ";" and such that i is the first such index. Another way to say it is that i is the position of the first separator if **there exists** k such that $i = k \times \ell$ and $H_{i,\ldots,i+\ell-1}$ and **for every** $j \in \mathbb{N}$, if $j < i$ then $H_{j,\ldots,j+\ell-1}$ does *not* encode ";". This can be captured using the operators allowed in a quantified mixed statetment and the $\forall$ and $\exists$ quantifiers.

2. We can similarly check that the last configuration is halting. Extracting the position i that encodes the last separator can be done in a way analogous to that of extracting the first one.

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

4. We can then write the condition that **for every** substring of H that has the form $\alpha ENC(;)\beta$ with $\alpha, \beta \in \{0,1\}^{\ell}$ and $ENC(;)$ being the encoding of the separator ";", it holds that $NEXT(\alpha, \beta)$ is true.

Together the above yields a computable procedure that maps every input $w \in \{0,1\}^*$ to $HALT_U$ into a quantified mixed statement φ_w such that $HALT_U(w) = 1$ if and only if $QMS(\varphi_w) = 1$. This reduces computing $HALT_U$ to computing QMS, and hence the uncomputability of $HALT_U$ implies the uncomputability of QMS. ■

10.4.2 Step 2: Reducing mixed statements to integer statements

We now show how to prove Theorem 10.5 using Theorem 10.8. 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 them by integers. 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 later 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 10.5, 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 the theorem 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 10.9 — Constructible prime sequence. There is a sequence of prime numbers $p_0 < p_1 < p_2 < \cdots$ such that there is a quantified integer statement $PCOORD(p,i)$ that is true if and only if $p = p_i$.

Using Lemma 10.9 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

$$\forall_{p\in\mathbb{N}} PCOORD(p,i) \Rightarrow DIVIDES(p,X) \tag{10.7}$$

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 10.5 is to prove Lemma 10.9, which we now proceed to do.

Proof. The sequence of prime numbers we consider is the following: We fix C to be a suficiently 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 $PCOORD(p,i)$ is simple:

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

We leave it to the reader to verify that $PCOORD(p,i)$ 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 10.8) implies the uncomputability of QIS, completing the proof of Theorem 10.5, and so also the proof of Gödel's Incompleteness Theorem for quantified integer statements (Theorem 10.4).

✓ **Lecture Recap**

- Uncomputable functions include also functions that seem to have nothing to do with NAND++ 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.

10.5 EXERCISES

R **Disclaimer** Most of the exercises have been written in the summer of 2018 and haven't yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 10.1 — Gödel's Theorem from uncomputability of QIS. Prove Theorem 10.4 using Theorem 10.5 ■

Exercise 10.2 — 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 10.3 — Expression for computing the index. Recall that in **??** asked you to prove that at iteration t of a NAND++ program the the variable `i` is equal to $t - r(r+1)$ if $t \leq (r+1)^2$ and equals $(r+2)(r+1)t$ otherwise, where $r = \lfloor \sqrt{t + 1/4} - 1/2 \rfloor$. Prove that there is a quantified integer statement $INDEX$ with parameters t, i such that $INDEX(t,i)$ is true if and i is the value of `i` after t iterations. ■

Exercise 10.4 — Expression for computing the previous line. Give the following quantified integer expressions:

1. $MOD(a,b,c)$ which is true if and only if $b = a \mod c$. Note if a program has s lines then the line executed at step t is equal to $t \mod s$.

2. Suppose that P is the three line NAND program listed below. Give a quantified integer statement $LAST(n,t,t')$ such that $LAST(t,t')$ is true if and only if $t' - n$ is the largest step smaller than $t - n$ in which the variable on the righthand side of the line executed at step $t - n$ is written to. If this variable is an input variable `x_i` then let $LAST(n,t,t')$ to be true if the current index location equals t' and

$t' < n$. ■

```
y_0    := foo_i  NAND foo_i
foo_i  := x_i NAND x_i
loop := validx_i NAND validx_i
```

Exercise 10.5 — 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 $I_0, \ldots, I_{m-1}$ with $I_j : (\{0,1\}^*)^{k_j} \rightarrow \{0,1\}^*$ such that a proof $(s_1, \ldots, s_n)$ that s_n is true is valid if for every i, either $s_i \in A$ or is some $j \in [m]$ and are $i_1, \ldots, i_{k_j} < i$ such that $s_i = I_j(s_{i_1}, \ldots, i_{k_j})$. 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 $F(x) \neq 0$. ■

7

[7] TODO: Maybe add an exercise to give a MIS that corresponds to any regular expression.

10.6 BIBLIOGRAPHICAL NOTES

10.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

10.8 ACKNOWLEDGEMENTS

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.)

III

EFFICIENT ALGORITHMS