

Gödel Coding

We turn symbols, formulas, and even proofs into natural numbers. We will then outline the proof of the First Incompleteness Theorem.

1. Coding symbols, formulas, and proofs

1.1. Motivation

The goal is to encode each formula as a natural number so that it can be processed mathematically and algorithmically. A useful analogy is ASCII, which encodes each character as a natural number ('A': 65, 'B': 66, ...). To encode a text (a string of characters), we simply concatenate the natural numbers ('BAC': 66/65/67). Since a computer does not encode natural numbers using the digits 0 through 9 but only the symbols 0 and 1, our ASCII characters are written using 8 bits:

A : 01000001
B : 01000010
Z : 01011010

For a string of characters, we simply concatenate the code of each character into a single natural number.

'BAC' : 010000100100000101000011

Decoding is easy: first divide the code into blocks of 8 bits.

There is an important distinction between a number and a representation of that number:

7, VII, seven, 111_2 , SSSSSS(0)

are different representations of the same natural number denoted by '7' (here in Arabic numerals, Roman numerals, words, binary notation, and the language $\mathcal{L}_0$ of Peano arithmetic). This is a familiar distinction in computer science. There is a deep philosophical dimension to naming things, even though the name does not change the thing itself. Just as the word *table* does not have four legs, arithmetic symbols serve only as a formal medium for representing natural numbers.

What is truly fundamental below is the existence of an encoding and a decoding procedure. Their inner workings are not so important and vary from one book to another. In any case, we will see that this coding is utterly unusable in practice.

1.2. Coding symbols

Consider the first-order language $\mathcal{L}_0 = (0, S, +, \times)$ for Peano arithmetic. We must encode the symbols of the language, the connectives, the quantifiers, and the variables $(x_i)_{i \geq 1}$.

We choose (arbitrarily) the following coding (see Peter Smith's book):

$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$	$\forall$	$\exists$	$=$	$($	$)$	0	S	$+$	$\times$
1	3	5	7	9	11	13	15	17	19	21	23	25	27

x_1/x	x_2/y	x_3/z	x_4	$\dots$	x_i	$\dots$
2	4	6	8	$\dots$	$2i$	$\dots$

For example, the code of the symbol “ $\exists$ ” is $c = 13$. The symbol with code $c = 21$ is “0” (the zero constant of the language). The variable x_i is encoded by the natural number $2i$.

To decode a code, simply read the table from bottom to top:

- if c is odd, find the symbol to which it corresponds;
- if c is even, the symbol is the variable $x_{c/2}$.

Remarks:

- For some proofs, we can limit the number of connectives, for example to $(\neg, \rightarrow, \forall)$ or to $(\neg, \vee, \forall)$.
- For decoding, it is important that no code be equal to 0.

1.3. Coding a formula

Consider a word in the language $\mathcal{L}_0$, that is, a finite sequence of symbols. It may be a formula “ $\forall S(x) = y$ ” or a meaningless sequence of symbols “ $S(\forall) \rightarrow \neg \times 0$ ”.

Let $c_1, c_2, c_3, \dots$ be the codes of the symbols in this word (read from left to right). These codes will be the exponents in a product of prime numbers:

$$2^{c_1} \times 3^{c_2} \times 5^{c_3} \times \dots$$

Let $\pi_1 = 2, \pi_2 = 3, \pi_3 = 5, \dots$ be the ordered sequence of prime numbers. We can now formalize our construction:

- Start with a word (or formula) $\mathcal{P}$, a sequence of symbols $a_1, a_2, \dots, a_l$ of $\mathcal{L}_0$.
- Associate its code c_i with each symbol a_i .
- The **Gödel number** $n(\mathcal{P})$ of the formula $\mathcal{P}$ is:

$$n = \prod_{i=1}^l \pi_i^{c_i}$$

Remarks. This is generally a very large number. The construction also applies to any word in $\mathcal{L}_0$, even if it is not a formula.

Example.

- $\mathcal{P} : “\neg(x = 0)”$
 - $[a_1, \dots, a_6] = [\neg, (, x, =, 0,)]$
 - $[c_1, \dots, c_6] = [1, 17, 2, 15, 21, 19]$
 - $n(\mathcal{P}) = 2^1 \cdot 3^{17} \cdot 5^2 \cdot 7^{15} \cdot 11^{21} \cdot 13^{19}$
- “2” does not exist in $\mathcal{L}_0$; its representation in $\mathcal{L}_0$ is “SS0”, whose Gödel number is $2^{23} \cdot 3^{23} \cdot 5^{21}$.

1.4. Decoding

Decoding is based on prime factorization (also called the fundamental theorem of arithmetic), and in particular on the uniqueness of this factorization.

Theorem 1.

For every natural number $n \geq 1$, there exist a natural number $l \geq 1$ and natural numbers $\alpha_1, \dots, \alpha_l \geq 0$ such that:

$$n = \pi_1^{\alpha_1} \times \pi_2^{\alpha_2} \times \dots \times \pi_l^{\alpha_l}$$

Moreover, the α_i are unique.

Thus, to decode a Gödel number n , that is, recover the encoded formula, we first factor n into primes. The exponents $c_1, \dots, c_l$ are the codes that recover the symbols of the formula one by one.

There is only one way to decode a number. Mathematically, this means that the coding is injective:

$$\text{If } n(\mathcal{P}) = n(\mathcal{Q}) \text{ then } \mathcal{P} = \mathcal{Q}.$$

Example.

$$n = 35\,872\,267\,500$$

- Prime factorization: $n = 2^2 \times 3^{15} \times 5^4$.
- The exponents give the codes $[2, 15, 4]$.
- Each code decodes into a symbol: $[x, =, y]$.
- Thus, the formula encoded by n is “ $x = y$ ”.

1.5. Coding proofs

We also want to encode an entire proof as a single natural number. Here, a proof is a finite sequence of formulas $(\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_L)$. How do we encode a proof? Each $\mathcal{P}_j$ is encoded by its Gödel number $n_j = n(\mathcal{P}_j)$. We then associate with the proof its **super-Gödel number**:

$$N = 2^{n_1} \cdot 3^{n_2} \dots \pi_L^{n_L}$$

that is:

$$N = \prod_{j=1}^L \pi_j^{n(\mathcal{P}_j)}$$

Remarks:

- This is a very, very large number. We will never compute it in practice.
- We will never mix Gödel numbers and super-Gödel numbers. They are two distinct encodings. We will know beforehand whether a given natural number denotes a Gödel number or a super-Gödel number.

How do we decode a super-Gödel number?

- Factor N into primes.
- Extract the list of exponents $(n_1, \dots, n_L)$.
- Decode each Gödel number n_j into a formula $\mathcal{P}_j$.
- This yields the list of formulas $(\mathcal{P}_1, \dots, \mathcal{P}_L)$.

2. Decoding algorithm

We have managed to encode symbols, formulas, and proofs as natural numbers. We will have to show that the corresponding operations on natural numbers are “sufficiently simple”, which will correspond to the notion of a *primitive recursive function* discussed in the following chapters. For now, we approach this issue algorithmically to prepare for the work ahead.

We begin with the encoding and decoding operations. For these operations to be “sufficiently simple” means that they can be programmed using only bounded “for” loops (the number of iterations is bounded as a function of the input). This means that we do not allow “while” loops for which we do not know in advance when the stopping condition will be reached.

2.1. Decoding a symbol code

It is easy to write an algorithm `decode_symbol_code(c)` that takes a natural number c as input (a valid symbol code) and returns the corresponding symbol $a \in \mathcal{L}_0$ (see the previous section). It does not even require a “for” loop.

Algorithm (`decode_symbol_code(c)`).

- – Input: a natural number c .
- – Output: a symbol a of $\mathcal{L}_0$.
- If $c = 1$, $a \leftarrow “\neg”$.
- If $c = 3$, $a \leftarrow “\wedge”$.
- ...
- If $c = 27$, $a \leftarrow “\times”$.
- If c is even, $a \leftarrow x_{c/2}$.
- Return a .

2.2. Decoding a Gödel number

Lemma 1.

Let $n = \pi_1^{\alpha_1} \times \cdots \times \pi_l^{\alpha_l}$ with $\alpha_i \geq 1$ ($i = 1, \dots, l$).

- $l \leq \log_2(n) \leq n$
- $\alpha_i \leq \log_2(n) \leq n$ ($i = 1, \dots, l$)

Proof. Recall that, by definition, $\log_2(2^n) = n$. Since $2^n \geq n$, we have $n \geq \log_2(n)$.

- $n = \pi_1^{\alpha_1} \cdots \pi_l^{\alpha_l} \geq \pi_1 \times \cdots \times \pi_l \geq 2^l$, so $\log_2(n) \geq l$.
- $n = \pi_1^{\alpha_1} \cdots \pi_l^{\alpha_l} \geq \pi_i^{\alpha_i} \geq 2^{\alpha_i}$, so $\log_2(n) \geq \alpha_i$.

□

The loop in the following algorithm is bounded by $K = \lfloor \log_2(n) \rfloor$. A meticulous reader who considers computing $\log_2(n)$ complicated may replace $\log_2(n)$ with n .

Algorithm (decode_godel_number(n)).

- – Input: a Gödel number n .
- – Output: a word/formula $[a_1, \dots, a_l]$.
- $\text{list} \leftarrow []$ (empty list).
- For k from 1 to $K = \lfloor \log_2(n) \rfloor$:
 - $c \leftarrow \text{exponent}(n, \pi_k)$
 - If $c > 0$, append $\text{decode_symbol_code}(c)$ to list .
- Return list .

We leave it to the reader to write the algorithm for decoding a super-Gödel number N .

2.3. Prime numbers

We allow the basic arithmetic operations $+$, $-$, $\times$, $/$. However, we must ensure that operations involving prime numbers are easy to perform.

First, the divisibility test $\text{is_divisible}(n, d)$, which decides whether d divides n , would be easy to program with a “while” loop. Here is the algorithm using a “for” loop.

Algorithm (is_divisible(n, d)).

- – Input: two natural numbers n, d .
- – Output: True iff d divides n .
- For i from 1 to n :
 - $n \leftarrow n - d$
 - If $n = 0$, return True (and end the loop).
 - If $n < 0$, return False (and end the loop).

From this we obtain an algorithm $\text{is_prime}(n)$ that returns True when n is prime and False otherwise, by testing whether there is a divisor d among $2, 3, \dots, n-1$.

Next, we can recover the exponent of p in the prime factorization of a natural number n . For example, $\text{exponent}(2^a 3^b 5^c, 3)$ is b .

Algorithm (exponent(n, p)).

- – Input: a natural number n and a prime number p .
- – Output: the exponent e of p in n .
- $e \leftarrow 0$.
- For k from 1 to $K = \lfloor \log_2(n) \rfloor$:
 - If $\text{is_divisible}(n, p)$:
 - $e \leftarrow e + 1$
 - $n \leftarrow n/p$
 - Otherwise, exit the loop.
- Return e .

Finally, we have used throughout the list $\pi_1 = 2, \pi_2 = 3, \dots$ of all prime numbers. We must verify that we can compute this list as far as needed. We program a function `next_prime(n)` that returns the first prime greater than n . Bertrand's postulate (proved by Chebyshev) states that there is always a prime between n and $2n$. It is therefore enough to test $n + 1, \dots, 2n$ for primality. We then easily obtain a function `prime_list(K)` that returns the list of all primes less than K .

3. Operations on formulas

We now justify that, given a Gödel number n , we can “easily” determine whether the corresponding formula has a given property. For example, is the formula well-formed? Is it a closed formula? Does it consist of a single variable? And for a super-Gödel number N : does it encode a correct proof?

3.1. Formulas

- `is_variable(n)` tests whether the formula with Gödel number n is one of the variables x_i .
- `is_term(n)` tests whether the expression with Gödel number n is a term of $\mathcal{L}_0$. Recall that the terms are the constant 0, the variables x_i , and, if t_1 and t_2 are terms, then $S(t_1)$, $t_1 + t_2$, and $t_1 \times t_2$ are also terms.
- `is_formula(n)` returns True when n encodes a valid formula of $\mathcal{L}_0$ (for example, “ $x = \forall$ ” is not valid).
- `is_closed_formula(n)` returns True if n encodes a formula with no free variables.

Here, for example, is the algorithm for testing whether n encodes a term of $\mathcal{L}_0$.

Algorithm `is_term(n)`:

- – Input: a natural number n .
- – Output: True iff n encodes a term of $\mathcal{L}_0$.
- $\mathcal{P} \leftarrow \text{decode_godel_number}(n)$
- If $\mathcal{P} = “0”$, return True.
- If $\mathcal{P}$ is a variable, return True.
- If $\mathcal{P}$ has the form “ $S(t)$ ”, return `is_term(t)`.
- If $\mathcal{P}$ has the form “ $t_1 + t_2$ ”, return `is_term(t_1) AND is_term(t_2)`.
- If $\mathcal{P}$ has the form “ $t_1 \times t_2$ ”, return `is_term(t_1) AND is_term(t_2)`.
- Otherwise, return False.

This is a recursive function whose depth is less than the length of $\mathcal{P}$, and hence less than $\log_2(n)$. We could write an algorithm for it using loops and stacks.

3.2. Proof

It is essential that we be able to decide efficiently whether a given proof is valid. This is the role of the function $\text{is_proof}(N, n)$.

- Let N be a super-Gödel number representing a sequence of formulas $\mathcal{P}_1, \dots, \mathcal{P}_L$.
- Let n be a Gödel number representing a formula $\mathcal{Q}$.
- The function $\text{is_proof}(N, n)$ returns True iff $\mathcal{P}_L = \mathcal{Q}$ and $\mathcal{P}_1, \dots, \mathcal{P}_L$ form a proof (a sequence of formulas obtained using the rules of inference).

$$\vdash \begin{array}{|l} \mathcal{P}_1 \\ \vdots \\ \mathcal{P}_{L-1} \\ \mathcal{P}_L = \mathcal{Q} \end{array}$$

Here it is easier to assume that proofs are written in a Hilbert-style system, with axioms and rules of inference of the form $\mathcal{P}_a \rightarrow \mathcal{P}_b$. There are no assumptions or temporary assumptions as in our natural deduction proofs. To obtain $\mathcal{P} \vdash \mathcal{Q}$, we actually prove $\mathcal{P} \rightarrow \mathcal{Q}$.

Thus, we can easily check that:

- either $\mathcal{P}_i$ is an axiom;
- or $\mathcal{P}_i$ follows from $\mathcal{P}_a$ and $\mathcal{P}_b$, where $a, b < i$ and $\mathcal{P}_a$ has the form $\mathcal{P}_b \rightarrow \mathcal{P}_i$.
- If $\mathcal{P}_1, \dots, \mathcal{P}_L$ form a valid proof and $\mathcal{P}_L = \mathcal{Q}$, return True.

3.3. Numerical value vs. numeral

We return to the difference between a number and its representation, with two new notations:

- $\ulcorner \mathcal{P} \urcorner$: the Gödel number of $\mathcal{P}$,
- the **numeral** $\bar{n}$: the representation of a natural number n in the language $\mathcal{L}_0$.

$\mathcal{P}$ formula of $\mathcal{L}_0$	$\longleftrightarrow$	$n = \ulcorner \mathcal{P} \urcorner$ Gödel number, in $\mathbb{N}$	$\longleftrightarrow$	$\bar{n} = \overline{\ulcorner \mathcal{P} \urcorner}$ numeral for n , in $\mathcal{L}_0$
---	-----------------------	--	-----------------------	--

Example:

$$\mathcal{P} : x = 0 \quad \longleftrightarrow \quad n = \ulcorner \mathcal{P} \urcorner = 2^{\dots} \cdot 3^{\dots} \cdot 5^{\dots} \quad \longleftrightarrow \quad \bar{n} = \overline{\ulcorner \mathcal{P} \urcorner} = \underbrace{SS \dots S}_{n \text{ occurrences}} 0$$

We will often write $\mathcal{Q}(n)$ instead of $\mathcal{Q}(\bar{n})$, since there can be no confusion. If we truly want an algorithm in the language $\mathcal{L}_0$, n must be written as the numeral $\bar{n} = SS \dots S0$.

3.4. Concatenation

Two formulas (or words) $\mathcal{P}$ and $\mathcal{Q}$ placed side by side give the **concatenation** $\mathcal{P} * \mathcal{Q}$. For example, “ $S(x)$ ” * “ $+$ ” * “ y ” gives “ $S(x) + y$ ”. This operation seems immediate, but it is more delicate to express when our formulas are represented by natural numbers:

- $\ulcorner S(x) \urcorner = 2^a \cdot 3^b \cdot 5^c \cdot 7^d$

- $\ulcorner + \urcorner = 2^e$
- $\ulcorner y \urcorner = 2^f$
- $\ulcorner S(x) + y \urcorner = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f$

Formally, if $n = \prod_{i=1}^l \pi_i^{\alpha_i}$ and $m = \prod_{j=1}^{l'} \pi_j^{\beta_j}$, then:

$$n * m = n \times \prod_{j=1}^{l'} \pi_{j+l}^{\beta_j}$$

3.5. Substitution

Consider a formula $\mathcal{P}$ with free variable x , usually denoted by $\mathcal{P}(x)$. When we write $\mathcal{P}(n)$, this is actually the substitution $\mathcal{P}[n/x]$. We must now express this in terms of Gödel numbers. We need a function $\text{substitution}(\ulcorner \mathcal{P} \urcorner, \ulcorner x \urcorner, n) : \mathbb{N}^3 \rightarrow \mathbb{N}$ that returns $\ulcorner \mathcal{P}(\bar{n}) \urcorner$, the Gödel number of the result of the substitution.

Example.

- $\mathcal{P}(x) : "y = x + 0"$, with Gödel number $\ulcorner \mathcal{P}(x) \urcorner = 2^{\dots} \cdot 3^{\dots} \cdot 5^{\dots} \cdot 7^{\dots} \cdot 11^{\dots}$.
- $\mathcal{P}(2) : "y = 2 + 0"$, but “2” is not a symbol of the language; we must write $\bar{2} = SS0$.
- Thus, $\mathcal{P}(\bar{2}) : "y = SS0 + 0"$ is indeed a formula of $\mathcal{L}_0$.
- $\ulcorner \mathcal{P}(\bar{2}) \urcorner = 2^{\dots} \cdot 3^{\dots} \cdot 5^{\dots} \cdot 7^{\dots} \cdot 11^{\dots} \cdot 13^{\dots} \cdot 17^{\dots}$ is the Gödel number of the result of the substitution.

Writing the substitution algorithm using concatenation is a good and not too difficult exercise.

4. Informal proof of the Incompleteness Theorem

4.1. Russell’s paradox

We will present the general idea behind the proof of the First Incompleteness Theorem. The proof relies on diagonalization, which here takes the form of self-reference, that is, a formula that speaks about itself. We begin with a short analogy involving Russell’s paradox, popularized as the following puzzle:

In a village, the barber shaves all the men who do not shave themselves, and only those men.

Who shaves the barber?

If we suppose that the barber does not shave himself, then the barber shaves himself. And if he shaves himself, then he must not shave himself. There is no way out!

Russell used the following version to justify that there can be no set containing all sets. If such a set $\mathcal{E}$ existed, we could consider:

$$y = \{x \in \mathcal{E} \mid x \notin x\}$$

Question: Is $y \in y$?

Remark: y is indeed a set in $\mathcal{E}$ because $\mathcal{E}$ contains all sets.

- If $y \in y$, then by the definition of y , $y \notin y$.
- If $y \notin y$, then by the definition of y , $y \in y$.

This is a contradiction, so the set of all sets cannot exist.

4.2. Diagonalization and the Gödel formula

Starting formula.

Here is the formula we will study:

$\mathcal{P}(x)$: “The formula obtained from the formula with Gödel number x , by substituting x for its free variable, is unprovable.”

Of course, in the following chapters we will write this formula in purely mathematical terms.

For now, observe that:

- A natural number n can be viewed as the Gödel number of a formula $\mathcal{Q}$ that we can recover (see the previous sections).
- If this formula $\mathcal{Q}$ has a free variable, for example $\mathcal{Q}(y)$, then we can perform the substitution $\mathcal{Q}[n/y]$, denoted by $\mathcal{Q}(n)$.
- Finally, for a given n , we can ask whether the closed formula $\mathcal{Q}(n)$ is provable. This is exactly what the formula $\mathcal{P}(n)$ does: “The formula $\mathcal{Q}(n)$ is unprovable.”

We must now justify that $\mathcal{P}(x)$ is indeed a formula of our language $\mathcal{L}_0$. We saw that “`is_formula()`” and “`is_proof()`” are simple algorithms, which means that they could be expressed in the language $\mathcal{L}_0$ of Peano arithmetic. But what about the assertion “this formula is provable”?

We can define:

$$\text{is_provable}(n) \quad \text{iff} \quad \exists N \text{ is_proof}(N, n).$$

The main problem, from an algorithmic viewpoint, is that we have no a priori bound on N , so a corresponding algorithm would use a “while” loop to test $N = 1, N = 2, \dots$. If the formula is provable, the algorithm eventually stops, but it does not if the formula is unprovable. It is therefore a bad algorithm from our point of view.

On the other hand, “ $\exists N \text{ is_proof}(N, n)$ ” is indeed a formula of our first-order language. Thus, “this formula is unprovable” is indeed a formula of $\mathcal{L}_0$. It follows that $\mathcal{P}(x)$, with x as a free variable, is a formula of $\mathcal{L}_0$.

Diagonalization.

Since $\mathcal{P}(x)$ is a formula of $\mathcal{L}_0$, it has a Gödel number. Let:

$$n = \ulcorner \mathcal{P}(x) \urcorner$$

be the Gödel number of the formula $\mathcal{P}$.

The Gödel formula is then defined by:

$$\mathcal{G} = \mathcal{P}(n)$$

It is therefore the formula $\mathcal{P}$ evaluated at its own Gödel number; in other words:

$$\mathcal{G} = \mathcal{P}(\ulcorner \mathcal{P} \urcorner)$$

The Incompleteness Theorem.

With this still informal formulation, here are the Incompleteness Theorem and its proof.

Theorem 2.

There exist true formulas that are unprovable.

We will show that $\mathcal{G}$ is such a formula. First, $\mathcal{G}$ says, “the formula obtained from the formula with Gödel number n , by substituting n for its free variable, is unprovable”. But “the formula obtained from the formula with Gödel number n , by substituting n for its free variable” denotes exactly the formula $\mathcal{P}$ evaluated at n , that is, $\mathcal{G} = \mathcal{P}(n)$, because n is the Gödel number of $\mathcal{P}$. Thus, the sentence $\mathcal{G}$ says exactly, “ $\mathcal{G}$ is unprovable”.

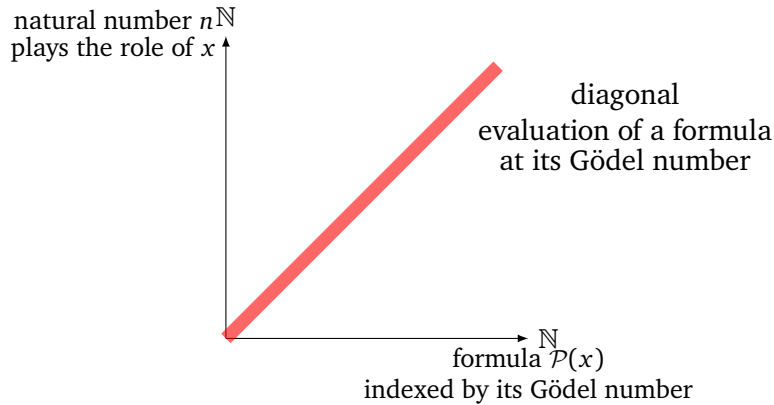
Proof.

- Suppose for a contradiction that $\mathcal{G}$ is false. Then $\neg\mathcal{G}$ is true, which means “ $\mathcal{G}$ is provable”. But by the Soundness Theorem, if $\mathcal{G}$ is provable, then $\mathcal{G}$ is true. This is a contradiction.
- Thus, $\mathcal{G}$ is true, but this means that “ $\mathcal{G}$ is unprovable”.
- Conclusion: $\mathcal{G}$ is true but unprovable.

□

4.3. Cantor’s diagonal

Why is the procedure explained above called diagonalization? The diagram below makes this easy to understand. We have formulas $\mathcal{P}$ (or $\mathcal{P}(x)$) that, as we have seen, are indexed by natural numbers in $\mathbb{N}$ (their Gödel numbers). We also have natural numbers $n \in \mathbb{N}$ that play the role of x in the evaluation. The Gödel sentence $\mathcal{G}$ is obtained by applying the formula $\mathcal{P}$ precisely at $n = \ulcorner \mathcal{P} \urcorner$.



Where does this idea of diagonalization come from? It is already present in Russell’s paradox, where the question “Is $x \in y$?” becomes, on the diagonal, “Is $y \in y$?”. Cantor famously used this method to prove that the set of real numbers is uncountable. Let us study his proof.

Theorem 3.

$\mathbb{R}$ is uncountable.

Proof. Suppose for a contradiction that the set of real numbers $[0, 1[$ is countable, that is, in bijection with $\mathbb{N}$. In more familiar terms, this means that all real numbers in $[0, 1[$ can be numbered $x_0, x_1, x_2, \dots$; that is, $[0, 1[= \{x_k\}_{k \in \mathbb{N}}$.

Write each real number in $[0, 1[$ in decimal notation, for example:

$$x_0 = 0.189\,335\dots$$

$$x_1 = 0.772\,004\dots$$

$$x_2 = 0.191\,542\dots$$

$$x_3 = 0.053\,477\dots$$

$\vdots$							
$\vdots$							
x_3		0	5	3	4	7 7	
x_2		1	9	1	5	4 2	
x_1		7	7	2	0	0 4	
x_0		1	8	9	3	3 5	
		Decimal digits					

$\vdots$						
$\vdots$						
		0	5	3	1	7 7
		1	9	2	5	4 2
		7	1	2	0	0 4
		2	8	9	3	3 5

- if this digit is 1, change it to 2;
- if this digit is not 1, change it to 1.

But by construction, x' cannot be any of the x_k , because x' and x_k have different digits in position k . This is a contradiction, since $\{x_k\}_{k \in \mathbb{N}}$ was supposed to enumerate all real numbers in $[0, 1[$. Therefore, $[0, 1[$ is uncountable, and consequently so is $\mathbb{R}$. $\square$