Recursion Theory Lecture Notes (Draft)

Ahmet Çevik

1 Background and origins

The goal of science is to make the algorithmic content of the world about us mathematically explicit. The tool we use to exploit the "principles" of the universe is the scientific method and reducing complex information to a simpler phenomenon. We seek patterns in the nature and observe fractal-like, spiral-like objects. We see objects, for example trees, and abstract them through their common properties. We solve a problem by reducing it to elements that are already known as an information. Mathematical information in particular may or may not have a structural content. Consider two infinite binary sequences

$A : 0101010101010101\ldots$
$B : 0100010101001110\ldots$

Now A seems to be more "structured" than B. In other words, B looks more "random" than A. This is because of the fact that the digits of A are periodic. It is an alternating 0-1 sequence. We can explain the information of A by an explanation simpler than A. However, if a sequence is random then we need at least n bits of axioms in order to determine its first n bits.

Theoretical limits of algorithmic information also determine the limits of computability. So then what are the theoretical obstacles to effective methods? What are the limits of solvability and decidability? What is unsolvable? These questions are investigated in a branch of mathematical logic called *recursion theory*, which is originated from the study of recursive (i.e., computable) functions.[1] One of its main aims is to study the algorithmic relationship between incomputable sets, functions, and relations. The term computable refers to "algorithmically computable".[2] We then must define what is meant by algorithmically computable.

The notion of "algorithm" or "effective computation" has been used throughout the history of mathematics. Euclid's algorithm for finding the *greatest common divisor* of given two natural numbers is known to be one of the first algorithms. The word "algorithm" was not used until the 9th century Persian mathematician al-Khwarizmi, from whom the term originates. In Latin, it was called "algorismus" when his works translated before the 12th century. Finally, the word "algorithm" was adopted in the 19th century English speaking world.

[1] Recursion theory is contemporarily called *computability theory* by many mathematical logicians. We will adopt both and use them interchangeably. For further reading on the computability and recursion terminology, see Soare (2016), *Turing Computability*, pp. 245-247.

[2] The term *algorithmically computable* is also known as *effectively computable*.

Here are some examples to algorithms:

- Finding the greatest common divisor of two natural numbers.
- Finding the prime factors of a positive integer.
- Sorting an unsorted finite list of numbers.
- Matrix multiplication.
- Finding the shortest path between two vertices in a graph.
- And many others...

What about things that cannot be an algorithm?

- Procedures whose description is infinite. For example,
  (1) Let x be the smallest prime number;
  (2) Add x with the least even number greater than 10;
  (3) Subtract the result of step 2 from the xth digit of π;
  (4)...
  (5)...

  We may allow loops in algorithms as long as its description is finite. For example,
  (1) Let $x = 1$;
  (2) Add x with the greatest prime number less than 100;
  (3) increment x by 1;
  (4) Go back to stage 2.
- Procedures that are not well-defined. For example,
  Let x be the coolest positive number smaller than 10.
- Procedures that cannot be completed in a finite amount of time. For instance,
  (1) Let $n = 0$;
  (2) If the number π contains three consecutive 7's in its decimal expansion, then let $x = n + 1$, otherwise let $x = n$.

Although Pascal invented the first calculator in the 16th century (and later improved as a general purpose machine by Charles Babbage in the 19th century), the concept of computation was profoundly used by Leibniz, who believed that principles of reasoning could be rigourously reduced to a formal symbolic system, a calculus of thought. He called this the *universal language (characteristica universalis)* in which, he believed, when a dispute arises, one just needs to calculate to find the solution.[3]

[3]Essentially, the very idea of reducing the knowledge about the reality into a structure of order goes back to Pythagoreanism, but we will not discuss this further.

Gottlob Frege was the first person who laid the foundations of formal logic since Aristotle. In his *Begriffsschrift*, Frege put Leibniz's idea into practice by introducing a system of logic, and improved this system in his later works.

Where does the concept of 'recursion' come into play? The term recursion, up until the early 1930's, meant 'defined by induction'. Roughly speaking, *recursion* is a process that calls itself using its sub-components as arguments. Best known example would be the definition of *Fibonacci sequence*:

$$F_0 = 0; \qquad F_1 = 1; \qquad \text{For } n > 1,\ F_n = F_{n-1} + F_{n-2}.$$

The characterization of recursion as a function type goes back to 19th century. For example, the recursive definition of addition and multiplication were provided by Grassmann (1861) and Peirce (1881) as

$$\begin{array}{l} \text{(i) } x + 0 = x \\ \text{(ii) } x + (y + 1) = (x + y) + 1 \end{array}$$

$$\begin{array}{l} \text{(i) } x \times 0 = 0 \\ \text{(ii) } x \times (y + 1) = (x \times y) + x. \end{array}$$

Dedekind (1888) showed that functions of this kind, defined in a recursive manner, constitute a unique class of functions. Peano (1889) then used Dedekind's definitions in axiomatizing arithmetic.

1.1 The foundational crisis of mathematics and Incompleteness.

Effective computability was never studied mathematically until the golden age of mathematical logic in the early 20th century mathematics. This was due to that an algorithm was considered to be a meta-mathematical object rather than a mathematical entity like a function, number, sequence, etc.[4] The urge to define what effective computability is came from the foundational crisis of mathematics. The crisis emerged from the debates about naive set theory. It was thought in naive set theory that for any property $\varphi(x)$ about sets,

$$A = \{x : \varphi(x)\}$$

would define a legitimate set.[5] This assumption was known as the *Axiom of Unrestricted Comprehension*, which later turned out to be false. Russell noticed in Frege's system that when one invokes this axiom in naive set theory and use Frege's abstraction principle, *Basic Law V*, it leads to a paradox. The paradox can be told as a story about a librarian compiling a catalogue of books. But let us instead look at the problem in terms of sets. Consider the formula $x \notin x$. We ask if

$$R = \{x : x \notin x\}$$

is a set. The answer is no. Because if R is a set, then either $R \in R$ or $R \notin R$. But if $R \in R$, then by its own definition, it cannot be in itself since it only contains sets which

[4] Same thing for the notion of *proof*.

[5] When we say a *property*, we mean any formula written in the language of set theory, that is, any predicate defined in the language of sets using the logical symbols and the membership $\in$ symbol.

are not members of themselves. However, if $R \notin R$, then R satisfies property $x \notin x$, so in this case it must be that $R \in R$. In either case, we get a contradiction. This is called *Russell's paradox*. The paradox caused a serious crisis in mathematics at a foundational level.

In the early 20th century, Hilbert had the idea of practicing mathematics using only "finitiary" arithmetic, referring to statements which use bounded quantifiers, proving facts with using such methods, etc. Of course, it wasn't really clear back then what was meant by "finitistic". In 1900, at the first International Congress of Mathematicians, Hilbert proposed 23 problems of mathematics which he hoped they would be solved until the end of millenium. The project emerged two themes: Computability theme and provability theme. In the computability theme, 10th problem was to find an algorithm to solve a given diophantine equation with integer coefficients. As a similar problem, in 1928, Hilbert and Ackermann introduced the *Entscheidungsproblem*, the problem of deciding whether or not the given first-order formula in the language of arithmetic is valid.

As for the provability theme, Hilbert proposed a long-term project for formalizing mathematics to achieve all mathematical knowledge in a complete and consistent set of axioms. A formal axiomatic system consists of a formal language, set of axioms, and rules of deduction. Hilbertian formal axiomatic system was expected to be consistent, complete, and decidable. But what is the use of such formal axiomatic system? If one could construct such theory, one would be able to do the following tasks.

1. Enumerating all possible sentences in the language of that theory.
2. Grammar checking. That is, deciding whether or not a given sentence is syntactically correct in the language of the theory.
3. Proof checking. Deciding whether or not a given proof is correct.
4. Effectively decide whether or not a statement has a proof.

In 1923, Skolem invented a formal system of arithmetic based on Hilbert's idea of finitary arithmetic using only the recursive definition of multiplication. Skolem's work was significant for three reasons:

1. Recursive functions are associated with effective computability for the first time.
2. Skolem talks about functions which are recursive (that is, computable) in the 'primitive' sense.
3. It was observed that these functions constitute a large part of ordinary mathematical functions.

Hilbert had the idea of defining a formal system for "all" of mathematics. Hilbert also wished to prove the consistency of arithmetic within the theory of arithmetic. He gave the famous radio address in 1930 announcing that there is nothing undecidable in mathematics, that questions for which we don't know the answer yet, will be answered in the future. Shortly after this radio address, Kurt Gödel proved a remarkable theorem

that there exist formally undecidable statements in arithmetic. He showed that there is a statement in formal arithmetic which is neither provable nor disprovable. He produced the liar-like sentence in formal arithmetic "This statement is false". Now if we can prove such a sentence in our formal system, then our system would prove something false and so the system will be inconsistent. But if not, then our system turns out to be incomplete, for that we have a true yet an unprovable statement.

Gödel numbering

Of course Gödel proved the existence of the foregoing self-referential statement in formal arithmetic. So we need to translate the self-referential statement "I am unprovable" into a sentence about numbers. We have to associate expressions of arithmetic with sentences of a formal axiomatic system if we want our system to be about numbers. We can encode every sentence by a natural number. For this we assign each symbol in the formal language to a code number and then obtain, in the end, a special code number for each sentence, and vice versa.

Consider Peano Arithmetic (PA) for our formal system. The language of PA, apart from the logical symbols, consists of the constant symbol 0, the successor function S, binary function symbols $+, \times$ and a binary relation symbol $<$. We also have possibly infinitely many $x, y, z, \ldots$ variable symbols. To capture the natural numbers greater than 0 we of course apply the successor function a number of times. For instance in the language of PA, we shall express 2 as $SS0$, or 3 as $SSS0$. Axioms of PA are as follows.

(i) 0 is a natural number.

(ii) For every natural number x, Sx is a natural number.

(iii) $\forall x \forall y(Sx = Sy \to x = y)$.

(iv) $\forall x \neg(0 = Sx)$.

(v) (Axiom Schema of Induction): $\{\varphi(0) \wedge \forall x(\varphi(x) \to \varphi(Sx))\} \to \forall x \varphi(x)$. Here the x variable in $\varphi(x)$ is *free*. That is, x is not bounded by any quantifier.

Axiom Schema of Induction consists of infinitely many instances. It is a kind of general form for each formula φ. Let us now code the symbols in the language of PA as follows. The table below is just an example, as the order can be changed.

$\neg$	$\wedge$	$\vee$	$\to$	$\iff$	$\forall$	$\exists$	$=$	(	)	0	S	$+$	$\times$	$<$
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29

x	y	z	$\ldots$
2	4	6	

It is possible to write countable infinitely many sentences using the language of PA. Some may be syntactically correct, some may be wrong. For example, $\forall x \exists y(Sx = y)$ is a syntactically correct sentences. Similarly, $1 + 1 = 1$ is as well, although not sound. However, $+1 <= 0$ is not a well-formed formula even though all symbols belong to the language of PA. It is easy to determine whether or not a given expression is a syntactically correct formula of PA. That is, grammar checking is straightforward. So we can separate well-formed formulas of PA from others.

We now assign a unique number to expressions which are formed by the symbols in the language of PA. Let e be an expression of the form $s_0 s_1 s_2 \cdots s_k$ containing $k+1$ many symbols. Let π_i denote the i-th prime number. The *Gödel number* of e is

$$\pi_0^{c_0} \cdot \pi_1^{c_1} \cdot \pi_2^{c_2} \cdot \ldots \cdot \pi_k^{c_k}$$

where c_i is the code of s_i. For instance the Gödel number of the expression $S0$ is $2^{23} \cdot 3^{21}$. Let us consider the formula $\exists y(S0 + y) = SS0$. This formula is syntactically correct and it actually means, under the standard interpretation, that there is a number, when added 1, equals to 2. The Gödel number of this formula is a massive number like

$$2^{13} \cdot 3^4 \cdot 5^{17} \cdot 7^{23} \cdot 11^{21} \cdot 13^{25} \cdot 17^4 \cdot 19^{19} \cdot 23^{15} \cdot 29^{23} \cdot 31^{23} \cdot 37^{21}$$

It is worth noting that for each sentence there is a unique Gödel code, and vice versa. That means, we can both compute the Gödel code of a given expression, and decode the Gödel code to obtain the corresponding expression. The uniqueness of this bijection is provided by the *Fundamental Theorem of Arithmetic:* Every integer greater than 1 is either prime itself or is the product of prime numbers, and that this product is unique, up to the order of the factor. Now that we coded the sentences of PA, it is possible to list them in a uniform way. The idea of listing expressions is essential in mathematical logic. We will use the same idea for listing algorithms.

Since we managed to code sentences, now we are interested in expressing their truth values via arithmetical relations. Given a formal language, consider the property of being a *formula* in that formal language. We shall define a relation for determining whether a given sentence is in fact a syntactically correct statement and denote it by Formula(n). We say that Formula(n) is true if and only if the number n codes a Gödel number for a syntactically correct sentence, i.e. a formula.

Apart from coding sentences, one can also code proofs. Formally speaking, a proof is a finite sequence of statements. If we can code every statement of the proof, then we can also code the proof itself following the same method of Gödel numbering, providing the prime numbers once again as the bases of exponents. Consider a sequence of statements $e_0, e_1, e_2, \ldots, e_n$. Suppose that this sequence constitutes a valid proof, meaning that every e_i is either an axiom or follows from at least one e_j such that $j < i$. To define the Gödel number of the proof

$$((e_0 \wedge e_1 \wedge \cdots \wedge e_{n-1}) \to e_n)$$

we first find the Gödel number of each statement e_i. Recall that in first-order logic, the inference $e_0, e_1, \cdots, e_{n-1} \therefore e_n$ is valid iff the sentence $((e_0 \wedge e_1 \wedge \ldots \wedge e_{n-1}) \to e_n)$ is a tautology.

Let us denote the Gödel number of e_i by g_i. We then define the Gödel number of the proof as

$$\pi_0^{g_0} \cdot \pi_1^{g_1} \cdot \pi_2^{g_2} \cdot \ldots \cdot \pi_n^{g_n}$$

Now we can define a binary relation Prf(m, n). We say that Prf(m, n) is true if and only if the expression with the Gödel number m is a *proof* of the statement with the Gödel number n.

Primitive recursive functions.

We need to ensure that the relations we have defined so far are effectively capturable inside Peano arithmetic, by which we mean the expressibility of predicates like *Formula, Prf*, and so on. This is where we need the concept of *computability*. For these predicates to be effectively captured inside our formal system, they should conform to Hilbert's finitary arithmetic. Functions we use in ordinary mathematics, such as addition, multiplication, exponentiation, the successor function, absolute value function, etc., are all fundamental to our mathematical studies. These simple functions are all calculable in the sense of Hilbert's finitary methods. In other words, they are all algorithmically computable. Moreover, they describe the simplest type of computability as they are all straightforwardly computable in a finite number of steps and that we are guaranteed to produce a result for any given argument. We will refer to such functions as *primitive recursive functions*. More formally, we define primitive recursive functions inductively as follows.

Definition 1. 1. The *initial functions* (a) - (c) are primitive recursive.

(a) The *zero function* is defined by

$$\mathbf{0}(n) = 0, \forall n \in \mathbb{N},$$

(b) The *successor function* is defined by

$$n' = n + 1, \forall n \in \mathbb{N},$$

(c) The *projection functions* U_i^k is defined by

$$U_i^k(\vec{m}) = m_i, \text{ for each } i = 1, \ldots, k \text{ and } k \geq 1 \text{ (where we write } \vec{m} = m_1, \ldots, m_k).$$

2. If $g, h, h_0, \ldots, h_l$ are primitive recursive, then so is f obtained from $g, h, h_0, \ldots, h_l$ by one of the rules:

(a) *Substitution*, given by:

$$f(\vec{m}) = g(h_0(\vec{m}), \ldots, h_l(\vec{m})),$$

(b) *Primitive recursion*, given by:

$$f(\vec{m}, 0) = g(\vec{m}),$$
$$f(\vec{m}, n+1) = h(\vec{m}, n, f(\vec{m}, n)).$$

The smallest class of functions which satisfy the definition is called *primitive recursive functions*. Many functions in ordinary mathematics is primitive recursive.

Example 1. We can show that + operation is primitive recursive. This is justified by the primitive recursion rule. That is,

$$m + 0 = m,$$
$$m + (n + 1) = (m + n) + 1 = (m + n)'.$$

Example 2. Multiplication operator $\times$ is primitive recursive.

Follows from the primitive recursion scheme:

$$m \times 0 = 0;$$
$$m \times (n+1) = (m \times n) + m.$$

Exercise 1. Show that the exponentiation function is primitive recursive.

Example 3. Show that the *predecessor* function defined by

$$\delta(m) = \begin{cases} m-1 & \text{if } m > 0 \\ 0 & \text{if } m = 0. \end{cases}$$

is primitive recursive.

It follows directly from the primitive recursive scheme
$\delta(0) = 0$,
$\delta(m+1) = m$

Example 4. Show that the *recursive difference* defined by

$$m \dot{-} n = \begin{cases} m-n & \text{if } m \geq n \\ 0 & \text{if } m < n. \end{cases}$$

is primitive recursive.

The primitive recursive scheme

$m \dot{-} 0 = m$,
$m \dot{-} (n+1) = \delta(m \dot{-} n)$,

together with the previous example gives the result.

Example 5 (Bounded sums)**.** If $f(\vec{m}, n)$ is primitive recursive, then $h(\vec{m}, p) = \sum_{n \leq p} f(\vec{m}, n)$ is primitive recursive.

Using the primitive recursive scheme,

$h(\vec{m}, 0) = f(\vec{m}, 0)$;
$h(\vec{m}, p+1) = \sum_{n \leq p} f(\vec{m}, n) + f(\vec{m}, p+1)$
$= h(\vec{m}, p) + f(\vec{m}, p+1)$

uses functions already known to be primitive recursive.

Exercise 2. Show that bounded product is primitive recursive.

Gödel proved that the predicates such as *Formula*, *Prf*, and a couple of other relations, are primitive recursive. The fact that these predicates are primitive recursive allows us to define the paradoxical Gödelian statement primitive recursively inside Peano arithmetic. For Gödel's theorems we give two propositions for which we shall omit the proof. But first we give a definition. We will revisit this definition later on when we define 'computability' more rigorously.

Definition 2. Let $A \subseteq \mathbb{N}$ be a set of natural numbers. If there exists an algorithm that determines whether or not $n \in A$ for any given $n \in \mathbb{N}$, then A is called a *recursive* (or *computable*) set.[6]

Theorem 1. For any computable set A, there exists a formula $\varphi(x)$ in the language of PA such that

$$n \in A \text{ if and only if } \varphi(n)$$

The n in the formula $\varphi(n)$ is of course described in the language of PA by repeatedly applying the successor function symbol S on the constant symbol 0, e.g., $S \cdots S0$. The reader should understand that any natural number is representated in the formal language of PA in this form. The theorem given above tells us that every computable set is captured by a formula in Peano arithmetic. Another required theorem is as follows.

Theorem 2. Every primitive recursive function is captured in PA. That is, if $f : \mathbb{N} \to \mathbb{N}$ is a primitive recursive function, then there exists a formula $\varphi(x, y)$ such that

$$\varphi(n, y) \text{ if and only if } f(n) = y$$

for every $n \in \mathbb{N}$.

Similar theorem also holds for n-ary relations. So all primitive recursive functions and relations can be captured in PA. In particular, the relations that Gödel used such as *Formula*, *Prf*, and *Diag*, which we will mention shortly, are all captured in PA.

Diagonalisation.

The next step of the theorem is to express the Gödealian sentence "This statement is unprovable" in the language of PA. For this we use a well known method called *diagonalisation*. Essentially, it is very similar to Cantor's method in proving that the set $\mathbb{R}$ of real numbers is uncountable. Suppose now that $\varphi(x)$ is a formula where x denotes the free variable of φ. Then, there is a Gödel number of this formula. Let us denote the Gödel number of φ by $\overline{\varphi}$. The diagonalisation method is the step where we substitute the Gödel number of φ into the free variable of φ. That is, the diagonalisation of $\varphi(x)$ will produce the formula $\varphi(\overline{\varphi})$. The reason that we can in fact produce this relies on the *Diagonal Lemma* we give below. Let us use the notation $\text{PA} \vdash \varphi$ to mean "φ is provable from PA".

Lemma 1 (Diagonal Lemma)**.** Suppose that $\psi(x)$ is a formula in the language of PA. Then, there exists a formula φ such that

$$\text{PA} \vdash \varphi \iff \psi(\overline{\varphi}).$$

Proof (Proof 1)**.** Let $\text{Diag}(\overline{\varphi(x)}) := \overline{\varphi(\overline{\varphi(x)})}$ be the *diagonal* function. Now this function is primitive recursive and so PA captures this function. Because it is a primitive recursive process to find the Gödel number of a given formula, and the inverse process. Secondly, substituting the Gödel number of a formula into the free variable is also primitive recursive

[6] The terms "computable", "recursive", "decidable" have the same meaning for sets as they can be used interchangably. For functions, we use "computable" or "recursive" rather than using the word "decidable".

by definition. Now this function maps the Gödel number of a formula with one free variable to the Gödel number of a sentence whose free variable is substituted by its own Gödel number. That is, from the formula $\varphi(x)$ we obtain the sentence $\varphi(\overline{\varphi})$. Then it suffices to define $\varphi = \psi(\mathrm{Diag}(\overline{\psi(\mathrm{Diag}(x))}))$. Then, in this case $\varphi = \psi(\overline{\varphi})$. Because when we compute the outward Diagonal function above, by the definition of Diag, we get the Gödel code of φ.

Proof (Proof 2)**.** Let $f : \mathbb{N} \to \mathbb{N}$ be the *diagonalization* function which maps the number $\overline{\phi(x)}$ to the number $\overline{\phi(\overline{\phi(x)})}$ and maps the rest of the elements in the domain to 0. As given above, f is a primitive recursive function. Then, the function f can be captured in PA, that is, for every $n \in \mathbb{N}$ there exists a formula $\theta(x, y)$ such that

$$\mathrm{PA} \vdash \theta(n, y) \iff f(n) = y.$$

Now let us define $\chi(x) := \exists y\,(\theta(x, y) \wedge \psi(y))$ and let φ be the formula $\chi(\overline{\chi(x)})$. This completes the proof of the lemma.

Now we will use the Diagonal Lemma to prove the First Incompleteness Theorem. It is inevitable that we omit some technical details in the proof, as Gödel's Incompleteness Theorem is normally taught as a one semester course when fully covered.

Originally, Gödel proved his theorem relying on a stronger assumption than what is actually sufficient. But we will see shortly that this does not change the presentation we give here.

Definition 3. Let T be a formal system. If for some formula φ in the language of T, the axioms of T separately proves each of the statements $\varphi(0), \varphi(1), \varphi(2), \ldots$, but T also proves $\exists x\, \neg\varphi(x)$, then T is called *ω-inconsistent*. If a system is not ω-inconsistent, then it is called *ω-consistent*.

Every inconsistent system is, by definition, ω-inconsistent. Also, every ω-consistent system is consistent, but not every consistent system needs to be ω-consistent.

We begin with using the Diagonal Lemma. Let us first consider the formula $\forall y\, \neg\mathrm{Prf}(y, x)$. Call the Gödel number of this formula m. This formula tells us that the formula whose Gödel number is x has no proof. Now let us diagonalise this formula and obtain the following sentence G.

$$G := \forall y\, \neg\mathrm{Prf}(y, m).$$

We have just obtained the desired paradoxical statement. G says that G has no proof. For reductio, suppose that G is provable. Let us denote the Gödel number of the proof by n. Then, $\mathrm{Prf}(n, m)$ must be true. Since we assumed that G was provable, we should be able to prove the statement $\forall y\, \neg\mathrm{Prf}(y, m)$. Since it holds for every y, it turns out that, in particular, $\neg\mathrm{Prf}(n, m)$ is provable, which is a contradiction.

Let us now suppose that $\neg G$ is provable. If PA is ω-consistent, then $\exists y\, \mathrm{Prf}(y, m)$ is provable from PA. In this case, for some n, $\mathrm{Prf}(n, m)$ is provable. But this gives us the proof of G, which is again a contradiction. Therefore, if PA is ω-consistent, we can neither prove G nor $\neg G$. This proves Gödel's First Incompleteness Theorem.

Barkley Rosser [**?**] managed to reduce the ω-consistency in the hypothesis of the theorem to plain consistency. Hence, Gödel's First Incompleteness Theorem can be stated as follows.

Theorem 3 (Gödel's First Incompleteness Theorem)**.** If PA is consistent, then it is not complete.

Are there natural examples of true mathematical statements which are not provable in PA or ZFC? We known that the Continuum Hypothesis cannot be settled in ZFC. As for PA, *Goodstein's Theorem* is one of the oldest theorems unprovable in PA. For a nice proof of its independence, see Adam Cichon's (1983) paper in the *Proceedings of American Mathematical Society*.

Goodstein's process is described as follows. Given a natural number N:

1. Write it in base x, i.e., write N as a sum of powers of x, and then the exponents also, the exponents of exponents, etc.
2. Increase the base of the representation by 1, then
3. Subtract 1 from the new number thus obtained.
4. Repeat the procedure 1-3, successively increasing the base by 1 and subtracting 1.

Example 6. For $x = 2$, $N = 25$, we get step 1:

$$25 = 2^{2^{2^1}} + 2^{2^1+2^0} + 2^0.$$

Then in step 2 we change the base to 3, which yields $3^{3^{3^1}} + 3^{3^1+3^0} + 3^0$. And in step 3 we subtract 1 from this number, which gives us $3^{3^3} + 3^{3+1} = 7625597485068$.

The process is said to *terminate* if the number 0 is eventually reached. *Goodstein's Theorem* says every Goodstein process terminates. It was shown by Kirby and Paris (1982) that this cannot be proved in PA (yet it is provable in ZFC).

Second Incompleteness Theorem

Next is to show that sufficiently strong formal systems cannot prove their own consistency. As a matter of fact, this is a consequence of the First Incompleteness Theorem. Define $\mathrm{Prov}(n) := \exists m\, \mathrm{Prf}(m, n)$. That is, $\mathrm{Prov}(n)$ holds if and only if the statement with Gödel number n is provable. We consider PA as our basis system. PA is a sufficiently strong system.[7] We may restate the First Incompleteness Theorem as

[7]Now that the predicate Prov has been defined, it is worth discussing what is meant by a *sufficiently strong* system. The criterion is that the system must be able to prove every true predicate which is, computability-wise, on a par with Prov. In computability theory, predicates that alike Prov have a special place. Gödel's theorems rely on two basic assumptions: (i) The first-order theory in consideration is expected to be ω-*consistent* (or consistent), (ii) if R is a primitive recursive function/relation, any statement of the form $\exists x\, R(x)$ is expected to be provable within the system whenever it is true. We call statements of the form $\exists x\, R(x)$, in recursion theory, Σ^0_1 statements. Now if (i) is not satisfied, then our system will simply be inconsistent. So, by *ex falso* rule, anything can be proved, hence the system would be complete in the way that we do not want. If (ii) is not satisfied on the other hand, since the system will not be able to capture the notion of provability, there is no point in talking about the "provability" of G. It only makes sense when the system understands what provability is. Consider a system T which is not sufficiently strong, for example any axiomatic system in propositional logic. Since no predicate on a par with Prov can be captured in T, the unprovability of G does not concern T. In fact, asking if G is provable within T would be non-sensical, similar to as we cannot speak about the truth of falsity of something when the system does not understand what the truth conditions are. Hence, sufficiently strong system means, in this context, that the system is able to prove every true Σ^0_1 statement. We call such systems Σ^0_1-*complete*. So we say that a system is *sufficiently strong* if and only if it is Σ^0_1-complete.

$$\mathrm{PA} \vdash G \iff \neg\mathrm{Prov}(\overline{G}).$$

We assume, of course, that PA is consistent. That is, we have $\mathrm{PA} \nvdash \bot$, where $\bot$ denotes a contradiction. Now let us define $\mathrm{Con}_{\mathrm{PA}} := \neg\mathrm{Prov}(\overline{\bot})$. Then, $\mathrm{Con}_{\mathrm{PA}}$ indirectly states that PA is consistent. Recall that the First Incompleteness Theorem says that if PA is consistent, then G is unprovable in PA. That is, $\mathrm{Con}_{\mathrm{PA}} \to \neg\mathrm{Prov}(\overline{G})$. In fact, we have

$$\mathrm{PA} \vdash \mathrm{Con}_{\mathrm{PA}} \to \neg\mathrm{Prov}(\overline{G}) \tag{*}$$

Moreover, in PA, we can prove

$$\mathrm{PA} \vdash G \iff \neg\mathrm{Prov}(\overline{G}) \tag{†}$$

For reductio, suppose that $\mathrm{PA} \vdash \mathrm{Con}_{\mathrm{PA}}$. Then, given that (*) holds, we get $\mathrm{PA} \vdash \neg\mathrm{Prov}(\overline{G})$. However, (†) says that $\neg\mathrm{Prov}(\overline{G})$ and G are in fact provably equivalent in PA. That is to say, one can be derived from the other. Hence, we have $\mathrm{PA} \vdash G$. But this contradicts the First Incompleteness Theorem and so our assumption must be false. We may then state the second theorem as follows.

Theorem 4 (Gödel's Second Incompleteness Theorem)**.** If PA is consistent, then PA cannot prove its own consistency.

Gödel's theorems are not limited to PA but they apply to any sufficiently strong system. As an immediate reaction so as to try to deny the theorem, even though it is futile, one may try to add G as an axiom to our formal system. Unfortunately this does not solve the problem. Even if we add G to our system, we automatically get a stronger system to which the theorem applies. We can again uniformly find another true-but-unprovable statement in this new system. No matter how large our set of axioms gets, we will never be able to avoid the incompleteness phenomenon. In some sense, not only did Gödel prove that sufficiently strong systems are incomplete, but he also proved that they are "incompletable".

One application of the Diagonal Lemma is Tarski's result on the *Undefinability of Truth* [**?**]. If M is a "structure" and φ is a statement which is "true" in M, then we denote this by $M \models \varphi$. Let us give a simple example to understand what we mean. Consider the statement

$$\forall x\, \exists y\, (y < x).$$

The statement above is false in the natural number "structure" due to the fact that there is no number in $\mathbb{N}$ strictly smaller than 0. On the other hand, the statement

$$\forall x\, \exists y\, (x < y)$$

is true in $\mathbb{N}$ since for every number in $\mathbb{N}$, there exists a larger number. We now give Tarski's theorem of the undefinability of truth.

Theorem 5 (Undefinability of Truth)**.** Let $T = \{\overline{\varphi} : \mathbb{N} \models \varphi\}$ be a set of natural numbers. Then, there exists no formula $\psi(x)$, in the language of arithmetic, such that $n \in T \iff \mathbb{N} \models \psi(n)$.

Proof. Suppose the contrary that there exists some formula $\psi(x)$ satisfying the condition of the theorem. From the Diagonal Lemma, it follows that there exists a formula φ such that PA proves the sentence

$$\varphi \iff \neg\psi(\overline{\varphi}).$$

Then, $\mathbb{N} \models \varphi$ if and only if $\mathbb{N} \models \neg\psi(\overline{\varphi})$ if and only if $\mathbb{N} \models \neg\varphi$. A contradiction.

Gödel's theorems should not be misinterpreted. The theorem does not say the followings.

(i) Mathematics cannot be formalised. (False)

Which fragment and what kind of mathematics do we refer to? It depends on whether we mean ordinary mathematics or "true" mathematics. Ordinary mathematics can be formalised, as ZFC set theory is a clear example of this formalisation. If we mean formalising "true" mathematics, then it will be chimerical. This is due to the fact that the self-referential Gödelian statement, uniformly obtained within the system, is a true arithmetical statement and that we should accept it as a part of the mathematical truth. What "has been proved" can be formalised, yet what is "being proved" or even "yet to be proved" cannot.

(ii) No formal system can be complete and consistent at the same time. (False)

Gödel's theorems do not apply to every formal system. For example, systems that are not Σ_1^0-complete (see footnote 7) can be simultaneously complete and consistent since such systems do not capture the Prov predicate, which is strong enough to compute all recursively enumerable predicates. An example of a complete and consistent system is the theory of dense linear orders without endpoints.[8]

(iii) Mathematics is inconsistent. (False)

The incompleteness theorem does not show that mathematics is inconsistent. It merely says that sufficiently strong consistent formal systems cannot prove their own consistency. This is not to say that our system is inconsistent. If it is consistent, however, this very fact cannot be proved within the system. Consider, for example, ZFC set theory. If ZFC is consistent, then its consistency cannot be proved within ZFC. Consider however a stronger system, call it ZFC^+. Now if ZFC^+ is consistent, it can prove the consistency of ZFC. But then the consistency of ZFC^+ cannot be proved within the same system. No matter how much we enlarge our system, a sufficiently strong system cannot prove its own consistency. This implies that it is impossible to prove absolute consistency results, but we can merely prove relative consistency, assuming the consistency of stronger systems.

(iv) Gödel's incompleteness theorem is false and not accepted by the mathematical community. (False)

The incompleteness theorem is in fact a legitimate theorem of mathematics. The only way to reject the incompleteness phenomenon is by denying the rules of classical logic. Although there may be mathematicians, particularly among non-logicians, who have not heard about the incompleteness phenomenon, those who know usually accept Gödel's theorems.

[8] See Hedman [**?**], §5.5, 2004.

2 Turing machines and Recursively Enumerable Sets

In the last section we defined the class of primitive recursive functions to potentially capture the intuitive notion of effective computability. However, we will show that primitive recursive functions are not suffcient to fully express what is intuitively 'computable' for reasons that will become clear in this section.

A 'computable' function which is not primitive recursive.

Although primitive recursive functions can express many arithmetical operations used in mathematics, it is not rich enough to capture *all* intuitively computable functions.

Theorem 6. There exists a computable function which is not primitive recursive.

Proof. When using primitive recursive functions, each derivation can be thought of as a finite string since it is a finite piece of information. Hence, we can effectively enumerate all primitive recursive functions. For this we use *Gödel numbering* which is standard method in logic for effectively enumerating a set of finite mathematical structures. Let f_n denote the n-th primitive recursive function. We define $g(x) = f_x(x) + 1$. We can deduce that g is a computable function but not primitive recursive since $g \neq f_x$ for all x. □

This argument is called *diagonalization* and is a widely used technique in recursion theory.

The *Ackermann function* is a concrete example of a recursive function but which is not primitive recursive. The Ackermann function is defined as:

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0. \\ A(x-1, A(x,y-1)) & \text{if otherwise.} \end{cases}$$

A common property of recursive functions that aren't primitive recursive is that they diagonalize against all primitive recursive functions. This is actually determined by how quickly the function grows compared to primitive recursive functions.

Definition 4. Let f and g be two functions from $\mathbb{N}$ to $\mathbb{N}$.

1. We say that g *dominates* f if for some $n_0 \in \mathbb{N}$

$$n > n_0 \to g(n) > f(n).$$

2. If S is a set of functions, we say that g dominates S if g dominates f for every $f \in S$.

An important fact to keep in mind is that if g dominates S, then $g \notin S$. We'll study the domination properties when we look at *high* and *low* sets in the later sections. But the point being here about the Ackermann function is that it dominates every primitive recursive function since it is 'superexponential', i.e. it grows faster than any exponential function.

A function is called *total* if it is defined on every argument. Otherwise, it is called *partial*. In fact, we must also take into consideration non-halting effective computations since they may still produce some important information during their computations. Non-halting effective computations have the nature of partial functions, i.e. functions that may be undefined on some arguments, because we may not be able to produce a final output for an arbitrarily given argument. For example, let $\psi(x) = \mu y\,[p(x, y) = 0]$ be a function, where $p(x, y)$ is some polynomial with integer coefficients and where $\mu x P(x)$ denotes "the least x such that $P(x)$ for some property $P(x)$". Then, ψ may be undefined for some values of x depending on the polynomial.

Now note that diagonalization method for partial recursive functions fails since $f(x)$ may be undefined for a partial function f. So we should consider partial functions for defining the intuitive notion of effective computability, for algorithms may not be defined on every argument as in the polynomial example we gave earlier.[9]

There have been different models of computation proposed which are believed to capture the class of intuitively computable functions. Kurt Gödel was the first logician who formally introduced *general recursive functions* in 1934 during his lectures at Princeton University. On the other hand, Alonzo Church [**?**] introduced his *lambda calculus* as a model of computation. Kleene, on the other hand, introduced μ-recursion to capture a larger class of computable functions. When we add the following rule to the class of primitive recursive functions, we obtain the class of *μ-recursive functions*.

Definition 5 (μ-operator)**.** Let $f(x) \downarrow$ denote that $f(x)$ is defined. If $g(\vec{n}, m)$ is a partial recursive function, then so is f given by

$$f(\vec{n}) = \mu m[g(\vec{n}, m) = 0],$$

where $\mu m[g(\vec{n}, m) = 0] = m_0 \iff g(\vec{n}, m_0) = 0$ and for all $m < m_0$, $g(\vec{n}, m) \downarrow \neq 0$.

So the μ-operator is a search operation. In fact, we have just expressed the search operation in the polynomial example here. If there is a solution, we will find the least such one. Otherwise, we will keep on searching unboundedly.

Turing machines

Alan Turing was perhaps the first person to describe a natural and universally accepted model of computation, called *Turing machine*, which is believed to correctly capture the notion of algorithmic computability. Turing machines were introduced in 1936 by Alan Turing [**?**] with the intention of understanding the mechanical process of a 'human idealised computer'.

For Turing, a human idealized computer has the following properties:

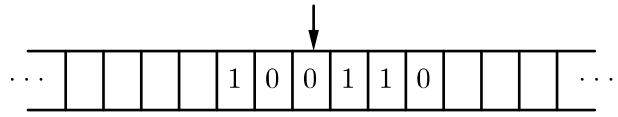
- Turing proposed a number of simple operations "so elementary that it is not easy to imagine them further subdivided".

[9] Note that Hilbert's tenth problem is, given a Diophantine equation with any number of unknown quantities and with integral coefficients, devising a process according to which it can be determined in a finite number of operations whether the equation is solvable in integers. This problem was shown to be unsolvable by a collection of works by Davis, Matiyasevich, Putnam and Robinson [**?**].

- He divided the work space into squares and he assumed it's one dimensional.
- He assumed finitely many symbols. Each square contains one symbol.
- He assumed finitely many internal states (of the human computer).
- The action of the machine is determined by the present state and the squares observed.
- The reading head examines one symbol at a time.
- It is assumed that the tape head moves only one square in either direction.

First let us give some notions from formal language theory. An *alphabet* is a finite set of symbols. A *string* is a finite sequence of symbols over some alphabet. Given two strings w and u, wu denotes the *concatenation* of w and u. The *length* of a string w is denoted by $|w|$. The *empty string* ϵ is the unique string of length 0. For any string w, $\epsilon w = w\epsilon = w$. Given an alphabet Σ, Σ^k denotes the set of strings of length k over the alphabet Σ. We denote the set of all strings over Σ by Σ^*. Note that since there is a one-to-one correspondence between Σ^* and $\mathbb{N}$, in the end we will be concerned with functions from $\mathbb{N}$ to $\mathbb{N}$.

We now define an abstract model of computation called *Turing machine* which basically consists of a control unit having finitely many states and an infinite tape on which we write symbols and which moves around the tape cells. The control unit carries the computation by following the given transition rules.



The formal definition is as follows.

Definition 6. A *Turing machine* M is a 5-tuple

$$(Q, \Sigma, \delta, q_0, q_f)$$

such that Q is a finite non-empty set of *states* where $q_0 \in Q$ is the *start state*, $q_f \in Q$ is the *halting state*, Σ is the alphabet, and δ is a partial function called the *transition function*, which is defined as

$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$$

such that L and R respective denote *left (L)* and *right (R)* movement of the tape head.

We may view δ as a set of quintuples. The interpretation is that if $(q, a, q', a', X) \in \delta$, then the machine M is in state q, reading the symbol a, and upon reading the symbol, the machines changes to state q', replaces a by a', and moves the tape head one square to the X direction. The map δ viewed as a finite set of quintuples is called a *Turing program*.

A *configuration* of a Turing machine is a triplet (x, q, y), where $q \in Q$, $x, y \in \Sigma^*$. We interpret a given configuration as follows: If $c = (x, q, y)$ is a configuration, we say that the Turing machine is in state q, the tape contains the string xy, and the tape head is reading the leftmost symbol of y. For convenience we may also write this configuration as xqy. In fact, we will adopt the latter notation.

To describe how the computation is carried out, let us say $x = w_1 a$ and $y = a_1 w_2$ such that $a, a_1 \in \Sigma$ and $w_1, w_2 \in \Sigma^*$. Therefore, c can be written as $w_1 a q_1 a_1 w_2$, and the transition function δ may define a transition rule from one configuration to another such that if $\delta(q_1, a_1) = (q_2, a_2, d)$ then a configuration $c = w_1 a q_1 a_1 w_2$ is transformed to, respectively for the left and right movement of the tape,

$$c' = w_1 q_2 a a_2 w_2 \quad \text{or} \quad c' = w_1 a a_2 q_2 w_2.$$

depending on which direction of the tape head moves $d \in \{L, R\}$.

If δ defines a transition from c to c', then we say that c *yields* c' and we denote this by $c \vdash c'$. Every such yield defines a *computational step*. The two exceptional cases $c = wq_1\epsilon$ and $c = \epsilon q_1 w$ can be handled by introducing the *blank symbol* # and so extending the definition of δ and replacing $wq_1\epsilon$ with $wq_1\#$, and replacing $\epsilon q_1 w$ with $\#q_1 w$.

A *computation* of a Turing machine with an input $w \in \Sigma^*$ is defined as a sequence of configurations $c_0, c_1, \ldots$ such that $c_0 = \epsilon q_0 w$ and $c_i \vdash c_{i+1}$ for each i. We say that the computation *halts* if the state symbol of some configuration c_i is q_f. In this case, we say that the machine *halts* on argument w and the output is whatever is written on the tape. We say that a Turing machine M *computes* a partial function ψ provided that $\psi(x) = y$ iff M with input x halts and yields output y. A function is *Turing computable* if there exists a Turing machine which computes it.

Example. We begin with a simple example. Suppose we work with functions from naturals to naturals, and assume we represent the number n by, say, n consecutive 1's in the tape. Let us define a Turing machine which computes the function $f(x) = x + 2$. Given x, represented by x consecutive 1's, we compute $x + 2$ by putting two more 1's to the end of the input string. Hence, the Turing program could be something as follows:

$$(q_0, 1, q_0, 1, R)$$
$$(q_0, \#, q_1, 1, R)$$
$$(q_1, \#, q_f, 1, R)$$

The computation of the machine on input 3 is as follows.

$$q_0 111\#\# \vdash 1 q_0 11\#\# \vdash 11 q_0 1\#\# \vdash 111 q_0\#\# \vdash 1111 q_1\# \vdash 11111 q_f$$

Example. Let us define a Turing machine, operating on finite binary strings, which replaces the last '0' in the input string (if there is any) with '1'.

The set of states can be put as $Q = \{q_a, q_f, q_0, q_1\}$. The alphabet is $\Sigma = \{0, 1\}$. Before defining the transition function, let us describe how the machine will operate intuitively.

We begin by reading the leftmost symbol of the input string, starting with the initial state q_0. We keep moving to the right and if we, at any point, encounter with the character '0', the machine will enter into a special 'memorise' stage q_a to reverse the tape head back

after we finish reading the entire input. When the tape head reverses back we will change the first encountered '0' to '1'. If in state q_a, we read the blank symbol, we reverse back and find the first '0'. If without entering the state q_a, we end up in state q_0 after we finish reading the string, it means we have no '0' in the input. In this case we terminate the computation. The Turing program then can be defined as follows.

$$\begin{array}{lll} (q_0, 1, q_0, 1, R), & (q_0, 0, q_a, 0, R), & (q_0, \#, q_f, \#, S), \\ (q_a, \#, q_1, \#, L), & (q_1, 1, q_1, 1, L), & (q_1, 0, q_f, 1, S), \\ (q_a, 1, q_a, 1, R), & (q_a, 0, q_a, 0, R). & \end{array}$$

Given n many inputs $x_1, \ldots, x_n$, we represent them as an input to a Turing machine by writing each as a block of x_k consecutive 1's and separating each block by the blank symbol #.

Exercise 3. Write a Turing machine for each of the following functions.

1. $f(x) = 0$.
2. $f(x) = 2x$.
3. $f(x, y) = x + y$.

Exercise 4. Define a Turing machine which takes a natural number in binary form and returns its successor in binary.

There are other variants of Turing machines, such as multiple tape Turing machines, one-way infinite tape Turing machines, non-deterministic Turing machines, probabilistic, quantum Turing machines, and register machines. They all compute the same class of functions. Hence, they are equivalent in computational power. One exception is *linear bounded machines*, where the tape size is limited to input size. These machines compute a smaller class of computable sets. There are also higher models which rely on computations over transfinite ordinals or real numbers. But they are out of scope of this course. For further reading we refer the reader to the book titled *Hypercomputation: Computing beyond the Church-Turing barrier* by Apostolos Syropoulos, or *Higher recursion theory* by Gerald Sacks. For a more computer scientific reference where you may find a discussion on the comparison between different variants of Turing machines, see *Elements of the theory of computation* by Lewis Papadimitriou.

An important remark we should make is that Turing machines and other models proposed by Kleene, Church, and Gödel are all equivalent. But among all, Turing machines are considered to be the most 'natural'. Hence, we usually regard Turing machines as the standard model of effective computability. The definition of effective computability relies on the following thesis.

Church-Turing Thesis: The class of Turing computable functions is exactly the class of effectively computable functions. That is,

$$\text{Effective computability} = \text{Turing computability}.$$

The hypothesis is not a mathematical statement because there is no mathematical definition of what's 'effectively' computable in the very general sense. On the other hand, Turing machines are well-defined mathematical objects. Therefore, the Church-Turing Thesis is a philosophical statement. From now on we will assume the Church-Turing Thesis and take it as *the* definition of effective computability.

Definition 7. A function $f : \mathbb{N} \to \mathbb{N}$ is called *partial recursive* if it is Turing computable. If f is defined on every argument then f is *total recursive* (or simply *recursive*).

Sets can be represented by their membership characteristic.

Definition 8. Let $S \subseteq \mathbb{N}$ be any set. The *characteristic function* of S is given by

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S. \end{cases}$$

We say that S is *recursive* (or *computable*) if χ_S is recursive.

Example 7. The following sets are recursive as we can describe, using the Church-Turing Thesis, an effective procedure for computing them.

(i) The set of natural numbers.

(ii) The set of even numbers.

(iii) $A = \{i : i \text{ is a prime number}\}$.

(iv) $A = \{(i, j) : i \text{ and } j \text{ are relatively prime}\}$.

(v) $A = \{i : i\text{th digit in the decimal expansion of } \pi \text{ is greater than } 5\}$.

(vi) $A = \{n : n \text{ is the Gödel number of a propositional tautology}\}$.

2.1 Basic results

Proposition 1. A set A is computable iff its complement, $\overline{A}$ is computable.

Proof. Since A is computable, we can decide whether or not $n \in A$ for any $n \in \mathbb{N}$. We see if $n \in A$. If so, then $n \notin \overline{A}$. Otherwise, $n \in \overline{A}$. □

First basic theorem tells us that there exists an effective way of enumerating all Turing machines (as Turing programs consist of finite sets of quintuples) and that there is a Turing machine called the *universal Turing machine* (UTM) which can simulate any other Turing machine.

Theorem 7 (Enumeration Theorem and Universal Machine)**.** Using Gödel numbering, there is an effective way of enumerating all partial recursive functions as $\varphi_1, \varphi_2, \varphi_3, \ldots$. That is, from index e we can computably obtain the Turing machine for computing the function φ_e. Using such a list, there exists a *universal Turing machine* which, for any given pair (e, x) of natural numbers, simulates the e-th Turing machine on argument x.

Proof. If we define the universal Turing machine as $\psi(e, x) = \varphi_e(x)$, then by Church-Turing Thesis there exists an index u for the universal machine such that $\varphi_u(e, x) = \psi(e, x)$. □

Notation. We will denote the e-th partial recursive function by φ_e or ψ_e. $\psi_e(x)\downarrow$ means ψ_e is defined (i.e., it halts) on input x. $\psi_e(x)\uparrow$ means ψ_e is undefined on x.

Next theorem tells us that there are (countably) infinitely many Turing machines which compute the same function.

Theorem 8 (Padding lemma)**.** For every $e \in \mathbb{N}$, there are infinitely many $i \in \mathbb{N}$ such that $\psi_e = \psi_i$.

Proof. For this we can take the e-th Turing machine description and we add dummy instructions which will give us another Turing machine description but computing the same function as before. Since we can add infinitely many different instructions we can find infinitely many indices. □

Another theorem about indices is called the *s-m-n theorem*, also known as the *parameter theorem*.

Theorem 9 (The *s-m-n* theorem)**.** Let $g(x, y)$ be a partial recursive funcion of two variables. Then there is a recursive function s of one variable such that, for all x, y,

$$\varphi_{s(x)}(y) = g(x, y)$$

Proof. Given a Turing machine M computing g and given a number $x \in \mathbb{N}$, we can define a Turing machine N that, on input y, simulates the action of writing the pair (x, y) on M's input tape and running M. We can then find an index $s(x)$ for the function computed by N. □

The intuition behind the *s-m-n* theorem is that data can be effectively incorporated into a program. Numbers can as well code partial recursive functions, and thus we may incorporate these numbers into a program as a subprogram. So the *s-m-n* theorem embodies the notion of subcomputation and an effective version of function composition. The theorem originates from Kleene and the name of the theorem comes from writing $s(x)$ in the more general form of $S_n^m(x_1, \ldots, x_k)$.

The next result is again one of the earliest theorems in recursion theory. It allows us to use an index for a partial recursive function that we are building in a construction as part of that very construction. Thus it forms the theoretical underpinning of the common programming practice of having a routine make recursive calls to itself. The following result is known as the *fixed point theorem*, but also known as the *recursion theorem*.

Theorem 10 (Recursion theorem)**.** For every recursive function f there exists some $n \in \mathbb{N}$ such that $\psi_{f(n)} = \psi_n$. Here, n is called the *fixed point* of f.

Proof. We first define the recursive "diagonal" function $d(u)$ by

$$\psi_{d(u)}(z) = \begin{cases} \psi_{\psi_u(u)}(z) & \text{if } \psi_u(u)\downarrow \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that d is total and one-to-one by the *s-m-n* theorem. Moreover, d is independent of f.

Given f, we choose an index j such that $\psi_j = f \circ d$ (here $\circ$ is the composition operator). We show that $n = d(j)$ is a fixed point of f. If f is total then so is $f \circ d = \psi_j$. Therefore, $\psi_j(j)$ will be defined and $\psi_{d(j)} = \psi_{\psi_j(j)}$. We have that

$$\psi_n = \psi_{d(j)} = \psi_{\psi_j(j)} = \psi_{f(d(j))} = \psi_{f(n)}.$$

The second equality follows from the definition of $\psi_{d(u)}(z)$, the third equality follows from that $\psi_j = f \circ d$, the fourth equality follows from our hypothesis that $n = d(j)$. $\square$

We ask if $\varphi_i(x) \downarrow$ is a computable relation of i and x. We will not give an answer to this question at the moment, but we can easily prove the following.

Proposition 2. If $\varphi_i(x) \downarrow$ is a computable relation of i and x, then so is the relation $\varphi_i(x) = y$.

Proof. Let y be the least z such that $\varphi_i(x) = z$. $\square$

Definition 9. We write $\varphi_{i,s}(x) = y$ if $i, x, y < s$ and $\varphi_i(x)$ outputs y in $< s$ steps. If such a y exists, then we say that $\varphi_{i,s}(x)$ *converges*, which we write as $\varphi_{i,s}(x) \downarrow$. Otherwise we say it *diverges* and we write as $\varphi_{i,s}(x) \uparrow$. Similarly, we write $\varphi_i(x) \downarrow$ if there exists some s such that $\varphi_{i,s}(x) \downarrow$.

From this definition, we can easily verify the following.

Corollary 1. (i) $\varphi_{i,s}(x) \downarrow$, $\varphi_{i,s}(x) = y$ are both computable relations of i, s, x, y.

(ii) $\varphi_i(x) \downarrow \iff \exists s \varphi_{i,s}(x) \downarrow$.

Example 8. Discuss the computability of

(i) $P = \{n :$ n-th linear equation with integer coefficients and with one variable has a solution greater than $n\}$.

Using Gödel numberings, we can list all such equations in order of the magnitude of coefficients or the length of the equation, and see if the n-th equation has a solution greater than n.

(ii) $Q = \{(n, i) :$ the first n digits of π contains i consecutive 7's$\}$.

Compute the first n digits of π and see if that initial segment contains i consecutive 7's.

(iii) Suppose that $Fib(x)$ holds iff x belongs to the Fibonacci sequence. Is $Fib(x)$ a decidable relation?

$Fib(x)$ is a decidable relation for that we can generate the Fibonacci numbers computably, and see if x is a member of this sequence. If we see some number $y > x$ without coming across x, then we know x won't be in the remaining part of the sequence.

(iv) $R = \{x :$ There exists a sequence of exactly x 7's in the decimal expansion of $\pi\}$.

This is more difficult. Unless we know more information about π we cannot compute R. We will show later how to prove the incomputability of a set.

Recursively enumerable sets

In Example 8 (iv), although we cannot decide whether $x \in R$ or not, π is what Alan Turing called a *computable real*. We can effectively write down the decimals of π as far as we like by using an infinite series for π known to coverge rapidly. So we can find the n-th digit of π given any natural number n. This will lead us to a new notion of effective enumeration, a kind of 'semi-decidability' for sets, and which is a more general form of computability.

Definition 10. A set A is called *recursively enumerable* (r.e.) (or *computably enumerable*) (c.e.) if there is an algorithm that enumerates the members of A. More formally, A is r.e. if A is the domain of some partial recursive function.[10] We denote the e-th r.e. set by

$$W_e = \text{dom}\varphi_e = \{x : \varphi_e(x) \downarrow\}.$$

Define $W_{e,s} = \text{dom}(\varphi_{e,s})$.[11]

We can think of an infinite r.e. set A as an infinite effective list of elements of A (but not necessarily in increasing or decreasing numerical order).

As we said the set $\{i : i \text{ is a prime number}\}$ is computable because there is an algorithm for deciding whether or not a number is prime. However, the set

$$S = \{i : \text{There are } i \text{ many consecutive 1's in the decimal expansion of } \pi\}.$$

is only 'semi-decidable' as we can only decide the membership of an element one way. That is, given $i \in \mathbb{N}$, we can answer positively when there are really i many consecutive 1's in the decimal expansion of π, but we may not always be able to answer negatively.

Concerning the relationship between recursive sets and recursively enumerable sets, we observe the following theorem.

Theorem 11. Every recursive set is recursively enumerable.

Proof. Let S be a recursive set. We define a recursively enumerable set R which will be equal to S. Initially let $R = \emptyset$. For every $n \in \mathbb{N}$, if $n \in S$, then enumerate n into R. Since $R = S$, S is r.e. □

The following theorem is another standard result about recursively enumerable sets saying that a set A is recursive if and only if there is an enumeration for A and for its complement $\overline{A}$.

Theorem 12 (Complementation Theorem)**.** A set A is recursive iff both A and $\overline{A}$ are recursively enumerable.

Proof. If A, hence $\overline{A}$, is recursive then both A and $\overline{A}$ are recursively enumerable. Now suppose that we have enumerations for A and $\overline{A}$. Then, for any given $n \in \mathbb{N}$, n is going to appear in the enumeration list of $\overline{A}$ if it is not going to appear in the enumeration list of A. Similarly, if n is not going to appear in the enumeration list of $\overline{A}$ then it must appear in the enumeration list of A at some point. Hence, we can decide for any given $n \in \mathbb{N}$ whether or not $n \in A$. □

[10] Alternatively, we will show in Proposition 5 that A is r.e. iff $A = \emptyset$ or A is the range of a partial recursive function.

[11] So if $x \in W_{e,s}$ then $x, e < s$. Note that $\varphi_e(x) = y$ iff $\exists s[\varphi_{e,s}(x) = y]$ and $x \in W_e$ iff $\exists s[x \in W_{e,s}]$.

Theorem 13 (Post, 1944)**.** Every infinite recursively enumerable set contains an infinite recursive subset.

Proof. Let S be an infinite recursively enumerable set. We define a computable set $A \subseteq S$ as follows. Initially let $A = \emptyset$. We then enumerate the members of S and whenever we find some n that we have not yet enumerated into A such that $n > m$ for every $m \in A$, we put n into A. Now A is the range of an increasing computable function. So A is recursive. □

Infinite sets which do not contain an infinite r.e. subset will be later called *immune* sets. It follows from Theorem 13 that a set is immune iff it does not contain infinite recursive subsets.

Recursively enumerable sets and closed under unions and intersections. Let $\langle x, y\rangle$ denote the standard pairing function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$.

Proposition 3. If A and B are recursively enumerable sets, then so is $A \cup B$.

Proof. Similar to earlier example, we take enumeration functions for A and B, say f and g, respectively. We define C to be the set of all pairs $\langle x, y\rangle$ such that $f(x)\downarrow$ or $g(x)\downarrow$. □

Exercise 5. Show that if A and B are recursively enumerable, then so is $A \cap B$.

Next, we give another characterization of r.e. sets. Let us first note the following equivalency.

$$e\text{-th Turing machine on input } x \text{ halts} \iff \varphi_e(x)\downarrow \iff x \in \mathrm{dom}\varphi_e.$$

Let us denote the $\mathrm{dom}\varphi_e$ by W_e, i.e. the e-th r.e. set. Similarly, denote $\mathrm{dom}(\varphi_{e,s})$ by $W_{e,s}$.

Definition 11. (i) If for all $x \in \mathbb{N}$ we have $x \in A \iff \exists y R(x, y)$ for some computable relation R, then A is called a Σ^0_1 set. We denote this by $A \in \Sigma^0_1$.

(ii) If for all $x \in \mathbb{N}$ we have $x \in A \iff \forall y R(x, y)$ for some computable relation R, then A is called a Π^0_1 set. We denote this by $A \in \Pi^0_1$.

(iii) If $A \in \Sigma^0_1$ and $A \in \Pi^0_1$, then A is a Δ^0_1 set, written as $A \in \Delta^0_1$.

Proposition 4. Let e be an index. Then, $\{x : \varphi_{e,s}(x)\downarrow\}$ is a Σ^0_1 set.

Proof. This immediately follows from the equivalency that $\varphi_{e,s}(x)\downarrow \iff \exists s \varphi_{e,s}(x)\downarrow$. □

Exercise 6. Show that $W_{e,s}$ is a computable set. Also observe that $W_e = \bigcup_{s\leq 0} W_{e,s}$.

Now we give the normal form theorem for r.e. sets.

Theorem 14 (Normal form theorem for r.e. sets)**.** The following statements are equivalent.

(i) A is an r.e. set.

(ii) A is a Σ^0_1 set.

(iii) $A = W_e$ for some $e \in \mathbb{N}$.

Proof. We start with $(i) \Rightarrow (ii)$. Assume that A is recursively enumerable. Then, there exists some e such that $A = \text{dom}(\varphi_e)$. Hence, $x \in A$ if and only if $\exists s \varphi_{e,s}(x) = y$. Then, $A \in \Sigma_1^0$.

Now we prove $(ii) \Rightarrow (iii)$. Suppose that $x \in A \Longleftrightarrow \exists s R(x, s)$, where R is a recursive relation. Define

$\psi(x) = 0$ if $\exists s R(x, s)$; otherwise leave it undefined. Now ψ is a partial recursive function since, by the Enumeration Theorem, $\psi = \varphi_e$ for some $e \in \mathbb{N}$. Then, $x \in A \Longleftrightarrow \psi(x) \downarrow \Longleftrightarrow \varphi_e(x) \downarrow$. Hence, $A = \text{dom}(\varphi_e)$. So $A = W_e$.

$(iii) \Rightarrow (i)$ is trivial. □

Definition 12. The *graph* of a (partial) function ψ is the relation

$$\text{graph}(\psi) = \{(x, y) : \psi(x) \downarrow = y\}.$$

Two partial functions are equal if their graphs are equal.

Example 9. It is easy to see that the following sets are r.e.

(i) $K_0 = \{\langle x, y\rangle \ : x \in W_e\} = \{\langle x, e\rangle : \exists s \exists y[\varphi_{e,s}(x) = y]\}$.

(ii) $K_1 = \{e : W_e \neq \emptyset\} = \{e : \exists s \exists x[x \in W_{e,s}]\}$.

(iii) $\text{range}(\varphi_e) = \{y : \exists s \exists x[\varphi_{e,s}(x) = y]\}$.

(iv) $\text{graph}(\varphi_e) = \{(x, y) : \exists s[\varphi_{e,s}(x) = y]\}$.

The following theorem justifies the intuitive description of an r.e. set A as one whose members can be effectively listed.

Proposition 5. A set A is recursively enumerable iff $A = \emptyset$ or A is the range of a recursive function.

Proof. $(\Leftarrow)$ If $A = \emptyset$, then A is r.e. Now suppose $A = \text{range}(f)$ for some recursive function f. Then A is r.e. by Example 9 (iii).

$(\Rightarrow)$ Let $A = W_e \neq \emptyset$. Choose any $a \in W_e$. Define the computable function f as

$$f(\langle s, x\rangle) = \begin{cases} x & \text{if } x \in W_{e,s+1} - W_{e,s} \\ a & \text{otherwise.} \end{cases}$$

Note that for $x \neq a$, each $x \in W_e$ is listed exactly once. Clearly, $A = \text{range}(f)$, because if $x \in W_e$, we choose the least s such that $x \in W_{e,s+1}$. Then, $f(\langle s, x\rangle) = x$ and so $x \in \text{range}(f)$. □

Theorem 15 (Uniformization Theorem)**.** If $R \subseteq \mathbb{N}^2$ is an r.e. relation, then there is a partial recursive function ψ such that

$$\psi(x) \downarrow \Longleftrightarrow \exists y R(x, y),$$

and in this case $(x, \psi(x)) \in R$.

Proof. Since R is r.e. and hence Σ_1^0, there is a computable relation S such that $R(x, y)$ holds if and only if $\exists z S(x, y, z)$. Define the partial function

$$\theta(x) = \mu u \ S(x, (u)_1, (u)_2)$$

and define $\psi(x) = (\theta(x))_1$. □

Proposition 6. A function is partial recursive iff its graph is recursively enumerable.

Definition 13. Two sets A and B are called *disjoint* if $A \cap B = \emptyset$. Given two disjoint sets A and B, a set C is called a *separating set* if $A \subseteq C$ and $B \cap C = \emptyset$. A and B are *recursively inseparable* if there is no recursive separating set. Otherwise, if there exists such a set, A and B called *recursively separable*.

Theorem 16. There exists recursively inseparable r.e. sets.

Proof. Define $A = \{e : \varphi_e(e) = 0\}$ and $B = \{e : \varphi_e(e) = 1\}$. Clearly, both are r.e. sets and disjoint, i.e. $A \cap B = \emptyset$. Suppose that there exists a recursive set C such that $A \subseteq C$ and $B \cap C = \emptyset$. Let χ_C denote the characteristic function of C. Since C is recursive, χ_C is total. By Enumeration Theorem there exists an index u such that $\chi_C = \varphi_u$.

Now u is either in C or not. If $u \in C$, then $\chi_C(u) = \varphi_u(u) = 1$. Then $u \in B$. But since $B \cap C = \emptyset$, we have that $u \notin C$. Contradiction.

Now suppose $u \notin C$. Then $\chi_C(u) = \varphi_u(u) = 0$. But then $u \in A$. But since $A \subseteq C$ in the hypothesis, $u \in C$. A contradiction. □

Recursively inseparable r.e. sets appear naturally. Sets of provable and refutable statements are recursively inseparable r.e. sets. That is,

$$P = \{gn(\varphi) : \varphi \text{ is provable in PA}\}$$

and

$$R = \{gn(\varphi) : \neg\varphi \text{ is provable in PA}\},$$

where $gn(\varphi)$ denotes the Gödel number of φ, are both recursively enumerable and recursively inseparable.

3 Incomputable sets

Gödel already showed the existence of undecidable statements in formal arithmetic. There is more to say about this however, for that not only do there exist undecidable statements, but there are uncountably many of them. In fact, the claim that not every function can be computable, and even that there are uncountably many non-computable functions, can be shown by a simple cardinality argument. There are only countably many Turing machine programs but uncountably many functions from $\mathbb{N}$ to $\mathbb{N}$. Therefore, there must be uncountably many incomputable functions $f : \mathbb{N} \to \mathbb{N}$.

We now describe the canonical example of a set which is recursively enumerable but not recursive. The corresponding decision problem is to decide whether or not a partial recursive function will ever be defined on a given argument. This is known as the *halting problem*, and its unsolvability may be seen as the main reason we have incompleteness.

Definition 14. Let $K = \{x : \varphi_x(x) \downarrow\}$ be the *halting set*.

Theorem 17. K is recursively enumerable.

Proof. K is the domain of the partial recursive function

$$\psi(x) = \begin{cases} x & \text{if } \varphi_x(x) \downarrow \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Now ψ is partial recursive by Church-Turing Thesis since $\psi(x)$ can be computed by applying the x-th partial recursive function to input x and giving output x only if $\varphi_x(x)$ converges. □

Theorem 18. K is not recursive.

Proof. Assume for a contradiction that K is recursive. If K had a recursive characteristic function χ_K, the following would also be recursive.

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } x \in K \\ 0 & \text{if } x \notin K. \end{cases}$$

But f cannot be recursive since $f \neq \varphi_x$ for every x. Therefore, K cannot be recursive. □

Combining the fact that K is a non-recursive r.e. set with the complementation theorem, we have the following.

Corollary 2. $\overline{K}$ is not recursively enumerable.

We usually prove the undecidability of a set by 'coding' it into other undecidable sets, such as the halting set.

Proposition 7. There is no algorithm to decide whether the domain of φ_x is empty.

Proof. We code the halting problem into the problem of deciding whether $\text{dom}(\varphi_x) = \emptyset$. That is, we show that *if* we could decide whether $\text{dom}(\varphi_x) = \emptyset$, *then* we could also decide whether φ_x halts on a given input. Define a partial computable function of two variables by

$$g(x, y) = \begin{cases} 1 & \text{if } \varphi_x(x) \downarrow \\ \uparrow & \text{if } \varphi_x(x) \uparrow. \end{cases}$$

Notice that g ignores its second input.

Via the *s-m-n* theorem, we can consider $g(x, y)$ as a computable collection of partial computable functions. That is, there is a computable function s such that, for all x, y,

$$\varphi_{s(x)}(y) = g(x, y).$$

Now,

$$\text{dom}(\varphi_{s(x)}) = \begin{cases} \mathbb{N} & \text{if } \varphi_x(x) \downarrow \\ \emptyset & \text{if } \varphi_x(x) \uparrow. \end{cases}$$

This is due to the fact that x is independent of y. By this way, if we could decide for a given x whether $\varphi_{s(x)}$ has empty domain, then we could solve the halting problem. □

Definition 15. An *index set* is a set A such that if $x \in A$ and $\varphi_x = \varphi_y$ then $y \in A$.

An index set can be thought of as coding a problem about computable functions (like the emptiness of domain problem) whose answer does not depend on the particular algorithm used to compute a function. We have the following result, which shows that non-trivial index sets are never computable. The proof is very similar to the proof of previous theorem.

Theorem 19 (Rice's Theorem)**.** An index set A is computable iff $A = \mathbb{N}$ or $A = \emptyset$.

Proof. ($\Leftarrow$) Trivial.

($\Rightarrow$) We prove by the contrapositive. Let $A \notin \{\emptyset, \mathbb{N}\}$ be an index set. Let e be such that $\text{dom}(\varphi_e) = \emptyset$. We may assume without loss of generality that $e \in \overline{A}$ (the case $e \in A$ being symmetric for that all functions in A compute the same thing). Fix $i \in A$. By the *s-m-n* theorem, there is a computable $s(x)$ such that, for all $y \in \mathbb{N}$,

$$\varphi_{s(x)}(y) = g(x, y) = \begin{cases} \varphi_i(y) & \text{if } \varphi_x(x) \downarrow \\ \uparrow & \text{if } \varphi_x(x) \uparrow. \end{cases}$$

If $\varphi_x(x) \downarrow$, then $\varphi_{s(x)} = \varphi_i$ and so $s(x) \in A$ since we know $i \in A$, while if $\varphi_x(x) \uparrow$, then $\varphi_{s(x)} = \varphi_e$ and so $s(x) \notin A$. Thus, if A were computable, then the halting set K would also be computable. Contradiction. Therefore, A is not computable. □

Theorem 20. K is not an index set.

Proof. For each n, let $f(n)$ be the index of $\{n\}$. That is, we let $\text{dom}\varphi_{f(n)} = \{n\}$. Then, by the Fixed Point Theorem, we get $W_{f(e)} = W_e$ for some e. Thus,

$$e \in W_e = W_{f(e)} = \{e\}.$$

But taking a different index e' of the set $\{e\}$ (using Padding Lemma), we have $e' \notin W_{e'} = \{e\}$. We then have $e \in K$ since $e \in W_e$, and $W_e = W_{e'}$ since we assumed e' is the index of the set $\{e\}$, but $e' \notin K$ since $e' \notin W_{e'}$. A contradiction. □

The following are some examples to index sets which correspond to natural unsolvable problems.

$K_1 = \{x : W_x \neq \emptyset\}$.
$\text{Fin} = \{x : W_x \text{ is finite}\}$.
$\text{Inf} = \{x : W_x \text{ is infinite}\}$.
$\text{Tot} = \{x : \varphi_x \text{ is total}\} = \{x : W_x = \mathbb{N}\}$.
$\text{Con} = \{x : \varphi_x \text{ is total and constant}\}$.
$\text{Cof} = \{x : W_x \text{ is cofinite}\}$.
$\text{Rec} = \{x : W_x \text{ is recursive}\}$.

Creative and productive sets

We want to look for more examples of incomputable sets. One example was introduced by Emil Post as follows.

Definition 16. A set P is *productive* if there exists a recursive function $\psi(x)$, called a *productive function* for P, such that

$$\forall x[W_x \subseteq P \Longrightarrow \psi(x) \in P - W_x].$$

An r.e. set C is *creative* if $\overline{C}$ is productive.

Example 10. Show that if A is creative, then it is not computable

Theorem 21. Creative sets exist. In particular, K is a creative set.

Proof. We know that K is recursively enumerable. We define the creative function for K to be the identity function $f : x \to x$. Assume that $W_e \subseteq \overline{K}$. Then, $e \notin W_e$ since otherwise we would have $e \in K \cap W_e$ which would be a contradiction. So $f(e) = e \in \overline{K} - W_e$. □

Note that $\overline{K}$ is productive with the identity productive function, i.e., $f(x) = x$.

No productive set A can be recursively enumerable, because whenever A contains every number in an r.e. set W_i, it contains other numbers, and moreover there is an effective procedure to produce an example of such a number from the index i. Similarly, no creative set can be decidable, because this would imply that its complement, a productive set, is recursively enumerable.

Proposition 8. If C is a creative set, then

(i) there exists an algorithm such that, given any n member of $\overline{C}$, one can effectively find $n + 1$ members of $\overline{C}$.

(ii) hence $\overline{C}$ contains an infinite r.e. subset.

Proof. (i) Let $y_1, y_2, \ldots, y_n$ be a (possibly empty) list of members of $\overline{C}$, and let C have a corresponding creative function f. Given n members for the list, there exists some i such that $W_i = \{y_1, y_2, \ldots, y_n\}$. Since $W_i \subset \overline{C}$, we have $f(i) \in \overline{C} - W_i$. But then $\{y_1, \ldots, y_n, f(i)\} \subset \overline{C}$, where $y_j \neq f(i)$ for $j \leq n$.

(ii) Enumerate an infinite r.e. subset $A = \{f(i_0), f(i_1), \ldots\}$ of $\overline{C}$ as follows

Find i_0 such that $W_{i_0} = \emptyset$, and enumerate i_0 into A.

Assume n numbers are enumerated into A already with corresponding index computes. Say $\{f(i_0), f(i_1), \ldots f(i_{n-1})\} = W_{i_n} \subset \overline{C}$. Enumerate $f(i_n)$ into A. □

Next we look at a natural way of comapring two sets in terms of how complex they are regarding their unsolvability.

Reducibilities

Definition 17. A set A is *many-to-one reducible* to B (written $A \leq_m B$) if there is a computable function f such that $x \in A$ iff $f(x) \in B$. We write $A \equiv_m B$ if $A \leq_m B$ and $B \leq_m A$.

If f is one-to-one, we say that A is *one-to-one reducible* to B (written $A \leq_1 B$). We write $A \equiv_1 B$ if $A \leq_1 B$ and $B \leq_1 A$.

Theorem 22. (i) If $B \leq_m A$ and A is recursive, then B is also recursive.

(ii) If $B \leq_m A$ and A is r.e., then B is also r.e.

(iii) A is r.e. iff $A \leq_m K$.

Proof. (i) Suppose $B \leq_m A$ via a recursive function f. Then $\chi_B = \chi_A \circ f$.

(ii) Suppose $B \leq_m A$ via a recursive function f. Use the Normal Form Theorem. Assume $A \in \Sigma_1^0$ with $x \in A \iff \exists y R(x, y)$ for some computable relation R. Then $x \in B \iff \exists y R(f(x), y)$, giving $B \in \Sigma_1^0$, hence being r.e.

(iii) Exercise. □

Definition 18. (i) A *computable permutation* is a one-to-one computable function from $\mathbb{N}$ onto $\mathbb{N}$.

(ii) A is *computably isomorphic* to B (written $A \equiv B$) if there is a computable permutation p such that $p(A) = B$.

Theorem 23 (Myhill's Isomorphism Theorem)**.** $A \equiv B$ if and only if $A \equiv_1 B$.

Proof. ($\Rightarrow$) Trivial.

($\Leftarrow$). Let $A \leq_1 B$ via f and $B \leq_1 A$ via g. Therefore,

$$\forall x \forall y [x \in A \iff f(x) \in B \text{ and } y \in B \iff g(y) \in A] \qquad (1)$$

We define a computable permutation h by stages $s \in \mathbb{N}$ so that $h(A) = B$. Suppose that by the end of stage $2s$ we have finite sets $X = \{x_1, x_2, \ldots, x_n\}$ on one side, and $Y = \{y_1, y_2, \ldots, y_n\}$ on the other side, and a one-to-one function h such that $h(x_i) = y_i$ for all $1 \leq i \leq n$ such that

$$\forall x \in X [x \in A \iff h(x) \in B] \qquad (2)$$

Stage $2s + 1$. We shall define $h(s)$ if it is not already defined. Compute $f(s) = t_1$. If $t_1 \notin Y$, define $h(s) = t_1$. If $t_1 \in Y$, say $t_1 = y_i$, then take $x_i = h^{-1}(t_1)$. Note that $x_i \in A$ iff $s \in A$ by (1) and (2). Compute $f(x_i) = t_2$. If $t_2 \notin Y$, define $h(s) = t_2$. Otherwise, $t_2 = y_j$ for some j. Take $x_j = h^{-1}(y_j)$ and note that $x_j \in A$ iff $s \in A$ as before. Compute $f(x_j) = t_3$. If $t_3 \notin Y$, define $h(s) = t_3$. Otherwise, $t_3 = y_i$ for some i. Take $x_i = h^{-1}(y_i)$ and note that $x_i \in A$ iff $s \in A$ as before. Compute $f(x_i) = t_4$. Continue in this fashion until a new element $z \notin Y$ is found and define $h(s) = z$. Note that $X \cup \{s\}$ has $n + 1$ elements but Y has only n elements, so the procedure must terminate.

Stage $2s + 2$. Find the value of $h^{-1}(s)$ in similar fashion using h^{-1} and g in place of h and f. □

Theorem 24 (Myhill, 1955)**.** (i) P is productive iff $\overline{K} \leq_1 P$.

(ii) C is creative iff $C \equiv K$ iff $A \leq_1 C$ for every r.e. set A.

4 Relative computability

Around the late 1930's, Alan Turing introduced *o-machines* in one part of his doctoral thesis and extended the definition of standard Turing machine. Using Turing's *o*-machines Emil Post, around 1944, used this to relativize computability and opened up a vast research area for recursion theorist. The idea is to relativize computability by taking the membership characteristic information of a set for granted and use it to compute other sets. While a set may not be computable, it may be 'computed' *relative* to another non-computable set, i.e., if we were given access to the membership information of another non-computable set. So the intuition is to use information concerning the membership of one set to help compute the membership problem of another set.

Let A and B be two sets. We want B to be computable from A if we can answer "Is $n \in B$?" using an algorithm whose computation uses finitely many queries about membership in A. That is, we ask finitely many questions whether some $m \in A$ during the computation of B. In other words, we consult A when computing B. In this case, A is called an *oracle* and the computation uses an oracle for A. We earlier said that sets are described by their characteristic functions, which define their *characteristic sequence*, that is, an infinite binary string coding the membership of natural numbers.[12] Thus an oracle for a set is just the characteristic sequence of that set. For this relativized form of computation, we use *oracle Turing machines*. An oracle Turing machine is just like a standard Turing machine with an extra *read-only* tape, called the *oracle tape*, on which the characteristic sequence of the oracle is written. Then, we can define the transition function as $\delta : Q \times \Sigma_1 \times \Sigma_2 \to Q \times \Sigma_2 \times \{L, R\}^2$, where Σ_1 denotes the oracle tape alphabet and Σ_2 denotes the work tape alphabet. In the computation of oracle Turing machines, we read the characteristic sequence of A written on the oracle tape and we perform the given instructions as usual. Since we use the information of an oracle in our computation, whatever we compute is only computable *relative* to that oracle.

We earlier said that Turing machine programs, i.e., hence partial computable functions, can be effectively listed. Recall that we denoted the e-th partial computable function by ψ_e. In that case, there was no use of an oracle. We now include oracles in the definition. We denote the e-th partial recursive function with an oracle A by Ψ_e^A. We also call this a *Turing functional* for reasons that will become clear shortly.

Let u be the total number of scanned non-empty cells in the oracle A during the computation. In this case, u is the maximum number used in the membership test of A. For convenience, then, we shall only 'use' the elements $x \leq u$. If no element is scanned, we let $u = 0$.

If a Turing functional Ψ_e^A is defined for a given argument and if this happens in $< s$ steps, and if $e, x, y, u < s$, then we write $\Psi_{e,s}^A(x) = y$ (or $\Psi_{e,s}^A(x) \downarrow = y$). If the Turing functional is defined with $\sigma \in 2^{<\mathbb{N}}$ on its oracle tape and $u \leq |\sigma|$ (therefore only σ is scanned), then we write $\Psi_e^\sigma(x) = y$.

Occasionally we also allow (total) functions f as oracles by defining Ψ_e^f to be Ψ_e^A where $A = \{\langle x, y\rangle : f(x) = y\}$, and allow partial functions as outputs.

Definition 19. (i) We say that a function f is *recursive in* A (or A-*recursive*), written

[12] With a little abuse of the notation, given a set A, we will write $A(n)$ to mean $\chi_A(n)$.

$f \leq_T A$, if there is some e such that $\Psi_e^A(x) \downarrow = y$ iff $f(x) = y$. A set B is said to be *A-recursive*, written as $B \leq_T A$, if χ_B is A-recursive. We write $B <_T A$ if $B \leq_T A$ and $A \not\leq_T B$.

(ii) We write W_e^A to denote $\mathrm{dom}\Psi_e^A$. If $B = W_e^A$ for some $e \in \mathbb{N}$, then we say that B is *recursively enumerable in A*.

We said that we could also call Ψ a *Turing functional* for that we have a more general description of a partial function since Ψ is a mapping from $2^{\mathbb{N}}$ to $2^{\mathbb{N}}$, as it takes, for instance, A to B if $\Psi_e^A = B$ for some e. We say that Ψ_e is *total* if $\Psi_e^A(x)$ is defined for every $A \subseteq \mathbb{N}$ and $x \in \mathbb{N}$, that is, it is total for all A there is some B such that $\Psi_e^A(x) = B(x)$ for every x.

Convention: The oracle Turing machine is self-delimiting in the sense that when a computation halts after reading σ on its oracle tape, then it must not read any more of the oracle tape and the machnine must turn off. By this property we have that if $\Psi_e^A(x) \downarrow = y$, then there exists a unique least string $\sigma \subset A$ such that $\Psi_e^\sigma(x)$ converges. Also, for every pair (σ, x) there can only be at most one y such that $\Psi_e^\sigma(x) = y$. From these conventions, we define the *oracle graph* of the Turing functional Ψ_e as the following r.e. set of *axioms*:

$$\{\langle \sigma, x, y\rangle : \Psi_e^\sigma(x) = y\}.$$

Theorem 25 (Use Principle)**.** The *use* conventions given above yield:

(i) $\Psi_e^A(x) = y \Longrightarrow \exists s\ \exists \sigma \subset A \left[\Psi_{e,s}^\sigma(x) = y\right]$,

(ii) $\Psi_{e,s}^\sigma(x) = y \Longrightarrow \forall t \geq s\ \forall \tau \supset \sigma \left[\Psi_{e,t}^\tau(x) = y\right]$,

(iii) $\Psi_e^\sigma(x) = y \Longrightarrow \forall A \supset \sigma \left[\Psi_e^A(x) = y\right]$.

This principle is important for later use. It implies that Ψ_e is continuous in the sense that the computation is progressive. The first item says that when a computation halts it does so in a finite number of stages and hence only a finite number of bits of the oracle tape can be scanned. The second item says that if a computation $\Psi_e^\sigma(x)$ is defined by the stage s, it will also be defined and give the same value for stages $t \geq s$ and for all extensions of σ. The third item says that if $\Psi_e^\sigma(x) \downarrow = y$ for some $\sigma \in 2^{<\mathbb{N}}$ then the computation is also defined for all infinite extensions of σ.

For convenience, we assume that for any string $\sigma \in 2^{<\mathbb{N}}$ and any $e, n \in \mathbb{N}$, $\Psi_e^\sigma(n)$ is not defined when $|\sigma| < n$. Hence if this computation converges, it does so in at most $|\sigma|$ steps.

All standard theorems now can be relativized. The following is the relativized enumeration and universal TM theorem .

Theorem 26. There is an effective enumeration of all oracle Turing machines, and a universal oracle Turing machine Ψ such that $\Psi^A(x, y) = \Psi_x^A(y)$ for all x, y and all $A \subseteq \mathbb{N}$.

Theorem 27. $B \leq_T A$ if and only if both B and $\overline{B}$ are r.e. in A.

The following is another known fact which easily follows from the relativization of the normal form theorem for r.e. sets. Note first the relativization of Σ_1^0 sets. We say that a set B is $\Sigma_1^{0,A}$ (or simply Σ_1^A) if $B = \{x : \exists y_1, \dots y_n R^A(x, y_1, \dots, y_n)\}$ for some A-recursive relation $R^A(x, y_1, \dots, y_n)$.

Theorem 28. The following are equivalent:

(i) B is r.e. in A.

(ii) $B = \emptyset$ or B is the range of some A-recursive total function.

(iii) B is $\Sigma_1^{0,A}$.

4.1 Turing degrees and the jump operator

Now we define the Turing degrees and the jump operator, both of which play a central role in computability theory.

Definition 20. (i) Let A and B be two sets. If $A \leq_T B$ and $B \leq_T A$, then we say that A and B are *Turing equivalent*, and this is denoted by $A \equiv_T B$.

(ii) We define the *Turing degree* (or *degree of unsolvability*) of a set $A \subseteq \mathbb{N}$ to be

$$\mathbf{a} = \deg(A) = \{X \subseteq \mathbb{N} : X \equiv_T A\}.$$

(iii) We write $\mathbf{D}$ for the collection of all such degrees, and define a partial ordering induced by $\leq_T$ on $\mathbf{D}$ by

$$\deg(B) \leq \deg(A) \iff B \leq_T A.$$

We write $\deg(A) < \deg(B)$ if $A <_T B$, i.e. if $A \leq_T B$ and $B \not\leq_T A$.

(iv) We denote Turing degrees by lowercase boldface Latin letters $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$.

Definition 21. (i) A degree $\mathbf{a}$ is called *recursively enumerable* if it contains a recursively enumerable set. We let $\mathbf{R}$ denote the set of all recursively enumerable degrees with the same ordering as for $\mathbf{D}$.

(ii) We say that a degree $\mathbf{a}$ is *recursively enumerable in* $\mathbf{b}$ if $\mathbf{a}$ contains some set A r.e. in some set $B \in \mathbf{b}$.

Intuitively, if two sets are of the same degree then they can be thought of as equally difficult to compute. If $\mathbf{a} < \mathbf{b}$, this means that sets of degree $\mathbf{b}$ are more difficult to compute than those of degree $\mathbf{a}$.

Definition 22. We define the *join* of two sets A and B as

$$A \oplus B = \{2i : i \in A\} \cup \{2i + 1 : i \in B\}.$$

The join $\mathbf{a} \cup \mathbf{b}$ of degrees $\mathbf{a} = \deg(A)$, $\mathbf{b} = \deg(B)$ is defined as

$$\mathbf{a} \cup \mathbf{b} = \deg(A \oplus B).$$

Definition 23. (i) A partially ordered set (poset) $\mathcal{L} = (L; \leq, \vee, \wedge)$ is called a *lattice* if any two elements have a least upper bound (also known as supremum, join, or union) and greatest lower bound (also known as infimum, meet, or intersection).[13] If a and b are elements of $\mathcal{L}$, $a \vee b$ denotes the least upper bound (l.u.b.) of a and b, and $a \wedge b$ denotes the greatest lower bound (g.l.b.). If $\mathcal{L}$ contains a least element and greatest element, these are called the *zero* element 0 and *unit* element 1, respectively. In such a lattice, a is the *complement* of b if $a \vee b = 1$ and $a \wedge b = 0$.

(ii) A poset closed under union but not necessarily under intersection is called an *upper semi-lattice*. A poset closed under intersection but not necessarily under union is called a *lower semi-lattice*.

Proposition 9. $\mathbf{a} \cup \mathbf{b}$ is the least upper bound of $\mathbf{a}$ and $\mathbf{b}$.

Proof. Exercise.

Basic structural properties of Turing degrees

The basic properties of the structure $(\mathbf{D}, \leq)$ can be given as follows.

Theorem 29. i) There is a least degree $\mathbf{0}$ which is the set of all recursive sets.

(ii) Each degree $\mathbf{a}$ has $\aleph_0$ elements.

(iii) The set of degrees $\leq \mathbf{a}$, for a given degree $\mathbf{a}$, is countable, i.e. $|\{\mathbf{b} : \mathbf{b} \leq \mathbf{a}\}| \leq \aleph_0$.

(iv) $\mathbf{D}$ has $2^{\aleph_0}$ elements.

Proof. Proof of (i) is obvious.

For (ii), let $\mathbf{a} = \deg(A)$ be a Turing degree. Then,

$$\begin{aligned}\mathbf{a} &= \{X : X \equiv_T A\} \\ &\subseteq \{X : X \leq_T A\} \\ &\subseteq \{\Psi_i^A : \Psi_i^A \text{ is total}\} \\ &\subseteq \{\Psi_i^A : i \geq 0\}.\end{aligned}$$

So $\mathbf{a}$ is a subset of a countable set and therefore $\mathbf{a}$ is countable. To show that $\mathbf{a}$ is also infinite, define

$$A_i = \begin{cases} A \cup \{i\} & \text{if } i \notin A \\ A - \{i\} & \text{if } i \in A. \end{cases}$$

Then for each i, j, we have $A_i(i) \neq A_j(i) = A(i)$, giving $A_i \neq A_j$. Moreover, for any i, $A_i \equiv_T A$. So $\mathbf{a}$ contains every A_i. Hence $\mathbf{a}$ is infinite.

Proof of (iii): Let $\mathbf{D}(\leq \mathbf{a}) = \{\mathbf{b} : \mathbf{b} < \mathbf{a}\}$, and let $A \in \mathbf{a}$. Then,

[13] Recall that a *partially ordered set* is a set that is ordered by a relation which is reflexive, antisymmetric and transitive.

$\mathbf{D}(\leq \mathbf{a})) = \{\deg(X) : X \leq_T A\} = \{\deg(\Psi_i^A) : \Psi_i^A \text{ is total}\} \subseteq \{\deg(\Psi_i^A) : i \geq 0\}$.

So $\mathbf{D}(\leq \mathbf{a})$ is countable.

For (iv), suppose for a contradiction that there are countably many Turing degrees. As proved in (ii), every degree contains countably many elements, so for any $X \subseteq \mathbb{N}$, $\deg(X) = \{A : A \equiv_T X\}$ is countable. If we assume that that there are countably many degrees, then this means the set of all subsets of $\mathbb{N}$ (of cardinality $2^{\aleph_0}$) is partitioned by Turing equivalence into countable union of countable equivalence classes. That is, $|\bigcup_{i\in I} C_i| = 2^{\aleph_0}$ for some countable set I. But we know that $2^{\mathbb{N}}$ is uncountable. Hence, I must be of size $2^{\aleph_0}$. □

For any $\mathbf{a}$ and $\mathbf{b}$ in $\mathbf{D}$, the least upper bound is their join. Therefore, the degree structure forms an upper semi-lattice. However, the greatest lower bound may not always exist for $\mathbf{D}$ or $\mathbf{R}$. Hence, neither $\mathbf{D}$ nor $\mathbf{R}$ forms a lattice. We will show later that $\mathbf{D}$ is strictly a semi-lattice.

One final remark about the Turing degree of functions. Turing degrees of functions are defined by the degree of their graphs. That is, $\deg(f)$ is defined as the degree of the graph of f.

Turing jump

Recall that the definition of the halting set K does not depend on any oracle (or it depends on $\emptyset$, if you prefer to view the partial recursive function in that definition as $\Psi_e^{\emptyset}$). We can relativize the halting set to any set $A \in \mathbb{N}$. This gives us what we call the 'Turing jump' of A, and it gives us a chance to study higher degrees in the Turing universe.

Definition 24. We define the *jump* of a set A to be

$$A' = K^A = \{x : \Psi_x^A(x)\downarrow\} = \{x : x \in W_x^A\}.$$

We pronounce A' as "A prime". The $(n+1)^{th}$ *jump* of A is defined as $A^{(n+1)} = (A^{(n)})'$, where $A^{(1)} = A'$.

Observe that $A \leq_T A'$. More generally, $A^{(n)} \leq_T A^{(n+1)}$. In fact, due to next theorem, that this relationship is strict. We can summarize some of the important properties of the jump operator as follows.

Theorem 30 (Jump Theorem)**.** Let $A, B \subseteq \mathbb{N}$. Then,

(i) A' is r.e. in A.

(ii) $A' \not\leq_T A$.

(iii) If A is r.e. in B and $B \leq_T C$ then A is r.e. in C.

(iv) A is r.e. in B iff $A \leq_1 B'$.

(v) $B \leq_T A$ iff $B' \leq_1 A'$.

(vi) $A \equiv_T B$ iff $A' \equiv_1 B'$.

Proof. (i) Note that K can be expressed more generally as $\{\langle x, y\rangle : x \in W_y\}$. We then have

$$\begin{array}{lll} \langle x, y\rangle \in A' & \iff & x \in W_y^A \\ & \iff & \exists s[x \in W_{y,s}^A]. \end{array}$$

The expression in square brackets is computable in A. So $A' \in \Sigma_1^A$, and so it is r.e. in A.

(ii) Follows from the fact that K is not recursive.

(iii) If $A \neq \emptyset$, then A is the range of some B-recursive function, and hence of some C-recursive function since $B \leq_T C$.

(iv) Follows from that A is r.e. iff $A \leq_1 K$.

(v) ($\Longrightarrow$) Suppose $B \leq_T A$. By (i) we have that B' is r.e. in B. Now since B' is r.e. in B and since we assume $B \leq_T A$, by (iii) it follows that B' is r.e. in A. Then, by (iv) we have $B' \leq_1 A'$.
($\Longleftarrow$) Assume now $B' \leq_1 A'$. Since both B and $\overline{B}$ are $\leq_1 B'$, we have that $B \leq_1 B' \leq_1 A'$ by our assumption. From this it follows that since $B \leq_1 A'$, by (iv) we have that B is r.e. in A. We get the same for $\overline{B}$. Hence both B and $\overline{B}$ are r.e. in A. Therefore, by the Complementation Theorem, $B \leq_T A$.

(vi) Follows immediately from (v). □

Let $\mathbf{a}' = \deg(A')$ for $A \in \mathbf{a}$. Note that $\mathbf{a}' > \mathbf{a}$ and $\mathbf{a}'$ is r.e. in (and above) $\mathbf{a}$. Let $\mathbf{0}^{(n)} = \deg(\emptyset^{(n)})$. Then, we have an infinite hierarchy of degrees

$$\mathbf{0} < \mathbf{0}' < \mathbf{0}'' < \cdots < \mathbf{0}^{(n)} < \cdots.$$

From the fact that the jump is strictly increasing, it follows that $\mathbf{D}$ has a least element but no maximal element. Note that $\mathbf{0}'$ is the degree of K which is Turing equivalent to $\emptyset'$.

We can give the following natural examples to the first few degrees. These sets were given earlier in the notes.

$\mathbf{0} = \deg(\emptyset)$ is the degree of all computable sets.
$\mathbf{0}' = \deg(\emptyset')$ is the degree of K, K_0, and other similar variants of K.
$\mathbf{0}'' = \deg(\emptyset'')$ is the degree of Fin, Tot, Inf
$\mathbf{0}''' = \deg(\emptyset''')$ is the degree of Cof, Rec.

4.2 Computable Approximations

Apart from recursive sets, on may also consider sets that to which we can approximated by a computable sequence by certain means. The methods one may use in the approximations here vary. One way of approximating to a recursively enumerable set A is to define a monotonic computable sequence of finite sets $\{A_s\}_{s\in\mathbb{N}}$ such that $A_s \subseteq A_{s+1}$, and let

$A = \bigcup_s A_s$. Consider now a more general approximation where a set is not approximated by the union of monotonic computable sequence of sets, but by $A = \lim_s A_s$ where membership of $x \in A$ can change finitely often as s tends to infinity. This is called a *limit computable* approximation, also called Δ_2 approximation. In some sense, we allow 'errors' in the computation where the value of $A(x)$ is changed finitely often in the approximation before it gets settled for good. We let $A \upharpoonright x$ denote $\{A(y) : y \leq x\}$.

Definition 25. (i) A set A is Σ_2 if there is a computable relation R such that

$$x \in A \Longleftrightarrow \exists y \forall z R(x, y, z).$$

(ii) A set A is Π_2 if $\overline{A}$ is Σ_2.

(iii) A set A is Δ_2 if $A \in \Sigma_2$ and $A \in \Pi_2$.

Definition 26. (i) A set A is *limit computable* if there is a computable sequence $\{A_s\}_{s\in\mathbb{N}}$ such that for all x,

$$A(x) = \lim_{s\to\infty} A_s(x).$$

By the Limit Lemma, we call $\{A_s\}_{s\in\mathbb{N}}$ a Δ_2-approximation for A.

(ii) Given $\{A_s\}_{s\in\mathbb{N}}$, any function $m(x)$ is called a *modulus of convergence* if

$$\forall x \forall s \geq m(x)\ [A \upharpoonright x = A_s \upharpoonright x].$$

We define the *least modulus* function as

$$m_A(x) = (\mu s)[A \upharpoonright x = A_s \upharpoonright x].$$

(iii) If A is an r.e. set, then a computable sequence $\{A_s\}_{s\in\mathbb{N}}$ is a Σ_1-approximation to A if $A = \bigcup_s A_s$ and $A_s \subseteq A_{s+1}$. In this case, $m_A(x)$ is a modulus and is called the *least modulus*. The least modulus is the first stage after which the approximation of $A(x)$ is always correct.

Theorem 31 (Limit lemma, Shoenfield 1959)**.** The following statements are equivalent.

(i) A is limit computable.

(ii) $A \in \Delta_2$.

(iii) $A \leq_T \emptyset'$.

Proof. (i)⇒(ii). Let $A(x) = \lim_s A_s(x)$ such that $\{A_s\}_{s\in\mathbb{N}}$ is a computable sequence. Then

$$x \in A \Longleftrightarrow \exists s \forall t[t \geq s \Rightarrow A_t(x) = 1]$$
$$x \in \overline{A} \Longleftrightarrow \exists s \forall t[t \geq s \Rightarrow A_t(x) = 0]$$

and hence $A \in \Sigma_2$ and $\overline{A} \in \Sigma_2$, so $A \in \Pi_2$. Therefore, $A \in \Delta_2$.

(ii)⇒(iii). Suppose that there are computable relations R and S such that

$$x \in A \Longleftrightarrow \exists s \forall t R(x, s, t) \ \text{ and } \ x \in \overline{A} \Longleftrightarrow \exists s \forall t S(x, s, t).$$

The predicate $\forall t R(x,s,t)$ is Π_1 and therefore computable in $\emptyset'$. Hence, the predicate $\exists s \forall t R(x,s,t)$ is Σ_1 in $\emptyset'$ and thus r.e. in $\emptyset'$, and similarly for $\exists s \forall t S(x,s,t)$. Therefore, A and $\overline{A}$ are both r.e. in $\emptyset'$. Hence, $A \leq_T \emptyset'$.

(iii)$\Rightarrow$(i). Let $\{K_s\}_{s\in\mathbb{N}}$ be a computable sequence such that $\bigcup_s K_s = K \equiv_T \emptyset'$. Assume $A = \Psi_e^K$. For every x and s define

$$f(x,s) = \begin{cases} \Psi_{e,s}^{K_s}(x) & \text{if defined;} \\ 0 & \text{otherwise.} \end{cases}$$

For every x, the first case holds for all but finitely many s. Therefore, $A(x) = \lim_s f(x,s)$. □

We imagine the degrees $\leq \mathbf{0}'$ forming an oval with $\mathbf{0}'$ on the top and $\mathbf{0}$ at the bottom. We first have the r.e. degrees, which forms a smaller inner oval. And then the Limit Lemma characterizes Δ_2 as the degrees in the outer oval containing the inner oval.

5 Arithmetical hierarchy

In this section we describe another way to classify the incomputability of sets according to the quantifier complexity of their definitions. We define the classes Σ_n^0, Π_n^0 (also shortly denoted as Σ_n, Π_n). The superscript 0 denotes that we are working in first order logic, that is we are quantifying over natural numbers. The subscript will denote the number of alternating quantifiers. For the second-order case, i.e. the analytical hierarchy, we refer the reader to Roger (1967) *Theory of recursive functions and effective computability* or Sacks (1990) *Higher Recursion Theory.*

Definition 27. (i) A set A is in $\Sigma_0^0 = \Pi_0^0 = \Delta_0^0$ if it is computable.

For $n \geq 1$:

(ii) A is in Σ_n^0 if there is a computable relation $R(x, y_1, \ldots, y_n)$ such that

$$x \in A \iff (\exists y_1)(\forall y_2)(\exists y_3)\ldots(Qy_n)R(x,y_1,\ldots,y_n),$$

where Q is $\exists$ for n odd, and $\forall$ for n even.

(iii) A is in Π_n^0 if there is a computable relation $R(x, y_1, \ldots, y_n)$ such that

$$x \in A \iff (\forall y_1)(\exists y_2)(\forall y_3)\ldots(Qy_n)R(x,y_1,\ldots,y_n),$$

where Q is $\forall$ for n odd, and $\exists$ for n even.

(iv) A is Δ_n^0 if $A \in \Sigma_n^0 \cap \Pi_n^0$.

(v) A set A is *arithmetical* if $A \in \bigcup_{n\in\mathbb{N}}(\Sigma_n^0 \cup \Pi_n^0)$.

Theorem 32. (i) $A \in \Sigma_n \iff \overline{A} \in \Pi_n$.

(ii) $A \in \Sigma_n$ (or Π_n) $\implies (\forall m > n)[A \in \Sigma_m \cap \Pi_m]$.

(iii) Sets in Σ_n (or Π_n) are closed under intersections and unions.

Proof. (i) If $A = \{x : (\exists y_1)(\forall y_2) \dots (Qy_n)R(x, y_1, \dots, y_n)\}$, then

$$\overline{A} = \{x : (\forall y_1)(\exists y_2) \dots (\overline{Q}y_n)\neg R(x, y_1, \dots, y_n)\}.$$

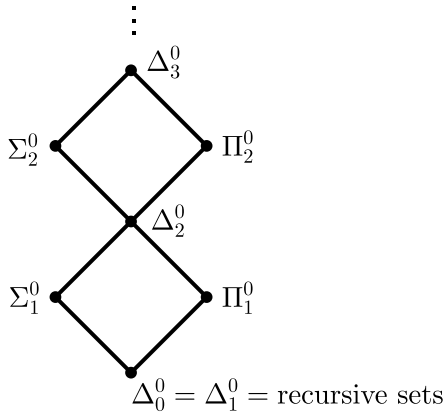
(ii) It can be generalized that, for example, if $A = \{x : (\exists y_1)(\forall y_2)R(x, y_1, y_2)\}$, then we can rewrite A by adding a 'dummy' quantifier as

$$A = \{x : (\exists y_1)(\forall y_2)(\exists y_3)[R(x, y_1, y_2) \ \wedge \ y_3 = y_3]\}.$$

The fact that $A \in \Pi^m$ can be shown similarly by adding a dummy $\forall$ quantifier to the beginning of the formula that defines A.

(iii) Exercise. □

Due to the fact that we can arbitrarily many dummy quantifiers, we have that $\Sigma_n \subseteq \Sigma_{n+1}$ (and similarly for Π_n). In fact, we will observe due to Post's Theorem that this subset relation is strict. Therefore, the arithmetical hierarchy does not collapse and we have the following strict relation as shown in figure below, where each line represents a strict subset relation.



Let us give some examples to determine the quantifier complexity of sets.

Example 11. Let $\text{Tot} = \{i : \varphi_i \text{ is a total function}\} = \{x : W_x = \mathbb{N}\}$. We can argue that Tot is a Π_2^0 set since

$$\begin{aligned} i \in \text{Tot} &\iff \forall n \varphi_i(n) \downarrow \\ &\iff \forall n \exists s[\varphi_{i,s}(n) \downarrow]. \end{aligned}$$

$$i \in \text{Tot} \iff \forall n \exists s \forall m[\varphi_{i,s}(n) \downarrow]$$

and that the expression in square brackets is computable, $\text{Tot} \in \Sigma_3^0 \cap \Pi_3^0$.

Example 12. We show that $\text{Fin} = \{e : W_e \text{ is finite}\}$ is in Σ_2.

$$\begin{aligned} e \in \text{Fin} &\iff \exists x \forall y \forall s \ [y \in W_{e,s} \to y < x] \\ &\iff \exists \forall \ [y \in W_{e,s} \to y < x] \end{aligned}$$

Since the relation in squared brackets is computable, Fin $\in \Sigma^0_2$.

Example 13. Let Rec $:= \{e : W_e \equiv_T \emptyset\}$. We show that Rec $\in \Sigma_3$.

$$\begin{array}{rcll} i \in \text{Rec} & \iff & W_x \text{ is computable} & \\ & \iff & \exists y[W_x = \overline{W}_y] & \text{(by complementation theorem)} \\ & \iff & \exists y[W_x \cap W_y = \emptyset \ \wedge \ W_x \cup W_y = \mathbb{N}] & \\ & \iff & \exists[\forall \ \wedge \ \forall\exists] & \text{(by Example 11)} \\ & \iff & \exists\forall\exists[\ldots]. & \end{array}$$

We will now show the relationship between the complexity classes of arithmetical hierarchy and the jump classes. The result is known as Post's Theorem in the literature, and it gives us some useful facts about the arithmetical hierarchy. First we need a lemma, for which we will omit the proof.

Lemma 2. For any $A \subseteq \mathbb{N}$, and $n \geq 0$,

$$A \in \Sigma^0_{n+1} \iff A \text{ is r.e. in } \emptyset^{(n)}.$$

Definition 28. A set A is Σ^0_n-*complete* if $A \in \Sigma^0_n$ and $B \leq_m A$ for every $B \in \Sigma^0_n$. Π^0_n-complete and Δ^0_n-complete sets are defined similarly.

Theorem 33 (Post's Theorem)**.** Let $A \subseteq \mathbb{N}$ and $n \geq 0$. Then:

(i) $\emptyset^{(n+1)}$ is Σ^0_{n+1}-complete.

(ii) $A \in \Sigma^0_{n+1} \iff A$ is r.e. in $\emptyset^{(n)}$

(iii) $A \in \Delta^0_{n+1} \iff A \leq_T \emptyset^{(n)}$.

Proof. (i) For each $A \subseteq \mathbb{N}$ we have

$$\begin{array}{rcl} A \in \Sigma^0_{n+1} & \iff & A \text{ is r.e. in } \emptyset^{(n)} \text{ (by Lemma 2)} \\ & \iff & A \leq_m (\emptyset^{(n)})' = \emptyset^{(n+1)} \text{ (by Jump Theorem (iv)).} \end{array}$$

In particular $\emptyset^{(n+1)} \in \Sigma^0_{(n+1)}$, and so by above it is $\Sigma_{(n+1)}$-complete since A is assumed to be Σ^0_{n+1}.

(ii) This follows from Lemma 2.

(iii) We have the following equivalencies

$$\begin{array}{rcl} A \in \Delta^0_{(n+1)} & \iff & A \in \Sigma^0_{(n+1)} \cap \Pi^0_{(n+1)} \\ & \iff & A \in \Sigma^0_{(n+1)} \text{ and } \overline{A} \in \Sigma^0_{(n+1)} \\ & \iff & A \text{ and } \overline{A} \text{ are r.e. in } \emptyset^{(n)} \text{ (by part (ii))} \\ & \iff & A \leq_T \emptyset^{(n)} \text{ (by relativized Complementation Theorem)} \end{array}$$

□

From Post's Theorem, the following corollary follows.

Corollary 3. For every $n > 0$, $\Delta_n \subset \Sigma_n$, $\Delta_n \subset \Pi_n$.

5.1 Domination properties and jump classes

We say that a function f dominates a function g if there exists some natural number m such that $f(n) \geq g(n)$ for all $n > m$. A partial function $\psi(x)$ dominates a partial function $\varphi(x)$ if ψ dominates φ whenever $\varphi(x)\downarrow$.

Definition 29. A set A is *computably dominated* if $g \leq_T A$ is dominated by a computable function.

Definition 30. The *least modulus* of A is

$$m_A(x) = \mu s[A_s \upharpoonright x = A \upharpoonright x]$$

Theorem 34 (Domination Properties)**.** Let $\{A_s\}_{s\in\mathbb{N}}$ be a computable enumeration of an r.e. set A and let f be a total function.

(i) If f dominates $m_A(x)$, then $A \leq_T f$.

(ii) For any $D \leq_T \emptyset'$,

$$D \equiv_T \emptyset' \iff (\exists f \leq_T D)[f \text{ dominates every partial recursive function}].$$

(iii) If $\{B_s\}_{s\in\mathbb{N}}$ is an enumeration of an r.e. set B and $m_A(x)$ dominates the least modulus function $m_B(x)$, then $B \leq_T A$.

Proof. (i) By definition, there is some y such that for all $x > y$, we have $\forall x[(x \in A) \iff x \in A_{f(x)}]$.

(ii) ($\Longrightarrow$) Build $f \leq_T \emptyset'$ by using $\emptyset'$ to determine for a given input x which $\psi_e(x)$ converges for $e \leq x$. Then define $f(x)$ exceed all these values.

($\Longleftarrow$) Follows from (i) because f dominates $m_K(x)$ (since D could be taken to be equivalent to $\emptyset' = K$ in the hypothesis).

(iii) By definition, there is some y such that for all $x > y$, we have $\forall x[(x \in B) \iff x \in B_{m_A(x)}]$. □

Definition 31. Let $a_0 < a_1 < a_2 < \ldots$ be a list of $A \subseteq \mathbb{N}$ in ascending order. Then the function p_A given by $p_A(x) = a_x$ is called the *principal function* for A. We say that an infinite set $E \subseteq \mathbb{N}$ is *hyperimmune* if p_E is not dominated by any computable function.

We know by Theorem 13 that every infinite r.e. set contains a recursive subset. An infinite set is called *immune* if it contains no infinite r.e. set. The intuition of immunity refers to the impossibility of computing an infinite subset of the set. Also note that every hyperimmune set is immune. The following proposition establishes a relation between computable domination and hyperimmune sets.

Proposition 10. A set A is not computably dominated iff there is a hyperimmune set $E \equiv_T A$.

Proof. ($\Longleftarrow$) is immediate since $p_E \leq_T E$.

($\Longrightarrow$): Suppose $g \leq_T A$ is not dominated by a computable function. Let $E = \text{ran}(h)$, where the function h is defined as follows: $h(0) = 0$, and for each n, let $h(2n+1) = h(2n) + g(n) + 1$ and let $h(2n+2) = h(2n+1) + p_A(n) + 1$. Clearly, $E \equiv_T h \equiv_T A$. Moreover $g(n) \leq h(2n+1)$, so that $h = p_E$ is not dominted by a computable function. □

Proposition 11. If $A \leq_T \emptyset'$ and A is non-recursive, then A is not computably dominated.

Definition 32. A degree which contains a hyperimmune set is called a *hyperimmune degree*. A degree which does not contain a hyperimmune set is called a *hyperimmune-free degree*.

Theorem 35 (Miller and Martin, 1968)**.** Every non-zero degree $\mathbf{d}$ which is comparable with $\mathbf{0'}$ is hyperimmune.

We will see later that hyperimmune-free degrees exist. These are degrees which are not comparable with $\mathbf{0'}$ and so they lay outside of the oval between $\mathbf{0}$ and $\mathbf{0'}$. In fact, due to Miller and Martin (1968), there are continuum many of them.

One of the earliest questions in computability theory was Post's problem of whether there exists an r.e. degree strictly between $\mathbf{0}$ and $\mathbf{0'}$. The answer is positive, but we are not ready to prove this yet. Knowing that there are other degrees between $\mathbf{0}$ and $\mathbf{0'}$, we can ask how close are they to $\mathbf{0}$ or $\mathbf{0'}$. The classification gives us the jump classes, which we shall look at it in this subsection. This will formalize the notion of sets being close to $\mathbf{0}$ or $\mathbf{0'}$ in terms of their algorithmic information content. Sets whose degree is 'close' to $\mathbf{0}$ have low information content, whereas sets whose degree is close to $\mathbf{0'}$ have high information content.

We know that the jump of $\mathbf{0}$ is $\mathbf{0'}$. Therefore, for degrees $\mathbf{a} \leq \mathbf{0'}$, $\mathbf{0'}$ is the least possible jump and $\mathbf{0''}$ is the greatest possible jump. We note that, $\mathbf{0}$ is the not only degree whose jump is $\mathbf{0'}$.

Definition 33. (i) A set $A \leq_T \emptyset'$ is called *low* if $A' \equiv_T \emptyset'$. It is called *high* if $A' \equiv_T \emptyset''$.

(iii) A set which is neither low nor high is called *intermediate*.

(iv) A degree is *low* if it contains a low set, it is *high* if it contains a high set.

(ii) Let $n \geq 1$. We say that A is low_n if $A^{(n)} \equiv_T \emptyset^{(n)}$. It is called high_n if $A^{(n)} \equiv_T \emptyset^{(n+1)}$. ($\text{low}_1$ means 'low' as defined above.)

Each low_n and high_n class is closed under Turing reducibility. That is, for low_n classes, we have a proper subset relation

$$\text{computable sets} \subset \text{low}_1 \subset \text{low}_2 \subset \ldots.$$

For a degree $\mathbf{a} \geq \mathbf{0}$, the range of the jump of $\mathbf{a}$ is $\mathbf{0'} \leq \mathbf{a'} \leq \mathbf{0''}$. Spector (1956) constructed a non-recursive Δ^0_2 low degree, hence gave the following result about the behavior of the jump operator.

Theorem 36 (Spector, 1956)**.** There exists a non-zero low degree. Hence, the jump operator is not one-to-one.

Theorem 37 (Sacks, 1963)**.** There exists a high degree $\mathbf{a} < \mathbf{0'}$.

The next theorem is known in the literature as *jump inversion* for Δ^0_2 degrees, and concerns the range of the Turing jump.

Theorem 38 (Friedberg, 1957)**.** If $\mathbf{b} \geq \mathbf{0'}$ then there exist a degree $\mathbf{a}$ such that $\mathbf{a'} = \mathbf{b}$.

So the jump operator is an *onto* function. A local version for this theorem for r.e. degrees is given by Shoenfield (1959).

Theorem 39 (Shoenfield, 1959)**.** If $\mathbf{a} \in \Sigma^0_2$ and $\mathbf{a} \geq \mathbf{0}'$ then there exists a degree $\mathbf{b} < \mathbf{0}'$ such that $\mathbf{b}' = \mathbf{a}$.

But let us now state a nice classification of high sets in terms of domination.

Theorem 40 (Martin's High Domination Theorem)**.** A set A is high iff there exists a function $f \leq_T A$ which dominates all recursive functions.

6 Construction methods

In this section we give some known results in classical degree theory. We will look at different construction methods. Although there are many results which use these construction methods, we aim to give some of the most important results for each method. We will start with the easiest method and move on to more sophisticated ones later.

6.1 Finite extension method

One of the earliest questions in computability theory was asked by Post: Does there exist an r.e. degree strictly between $\mathbf{0}$ and $\mathbf{0}'$? Although we will not answer this now, it is first natural to ask if $\mathbf{D}$ is totally ordered. This is answered negatively by Kleene and Post (1954).

Definition 34. Two degrees $\mathbf{a}, \mathbf{b}$ are called *incomparable* if $\mathbf{a} \not\leq \mathbf{b}$ and $\mathbf{b} \not\leq \mathbf{a}$.

We now show that there exist incomparable degrees by building sets A and B such that $A \not\leq_T B$ and $B \not\leq_T A$. Furthermore, we will show that their degrees are below $\mathbf{0}'$. The main idea is that instead of considering a single complicated condition like $A \not\leq_T B$, we shall consider an infinite sequence $\{R_e\}_{e\in\mathbb{N}}$ of simpler conditions. We call these conditions *requirements*. Here, each R_e will be defined as the condition $A \neq \Psi_e^B$. At each stage of the construction we build more of the characteristic sequence of the sets that we want to construct. For these sets, we respectively define strings σ_s and τ_s at stage s. In the end we define $A = \bigcup_{s\in\mathbb{N}} \sigma_s$ and $B = \bigcup_{s\in\mathbb{N}} \tau_s$. We use an oracle for $\emptyset'$, at each stage of the construction when choosing σ_s and τ_s, and we choose these values so as to ensure that the next stage condition in our list of requirements is satisfied. We also ensure that $\sigma_s \subset \sigma_{s+1}$ for each $s \in \mathbb{N}$. We call this method the *finite extension method* for it ensures that σ_{s+1} is a finite extension of σ_s for each s.

Theorem 41 (Kleene and Post, 1954)**.** There exist incomparable degrees below $\mathbf{0}'$.

Proof. We construct two sets A and B that are computable in $\emptyset'$ such that $A \not\leq_T B$ and $B \not\leq_T A$. We break these two requirement into infinite sequences of much simpler conditions and at each stage of the construction we aim to satisfy one. The requirements are as follows.

$$\begin{array}{lcl} R_{2e} & : & A \neq \Psi_e^B \\ R_{2e+1} & : & B \neq \Psi_e^A \end{array}$$

We use the finite extension method to construct $A = \bigcup_{s\in\mathbb{N}} \sigma_s$ and $B = \bigcup_{s\in\mathbb{N}} \tau_s$. We satisfy a single requirement at each stage and once it is satisfied it will remain satisfied forever. Let $\sigma_0 = \tau_0 = \emptyset$. Suppose that σ_s and τ_s are given at stage $s+1$.

If $s+1 = 2e$, then we satisfy R_e. Let $x \in \mathbb{N}$ be the first element such that $\sigma_s(x)$ is not defined yet. This means that we have not yet decided whether or not x should be in A. We decide this now and we use x to *witness* $A \neq \Psi_e^B$. In other words, we satisfy $A(x) \neq \Psi_e^B(x)$. That is, we make A on argument x different than Ψ_e^B. Since we have not constructed B yet, we do not know if $\Psi_e^B(x)$ converges, i.e. is defined. However, we do know that if it converges then there exists some $\tau \subset B$ such that $\Psi_e^\tau(x)$ is defined (by the Use Principle). Since $\tau_s \subseteq B$ by construction, if such a τ will exist then it will be compatible with τ_s because B extends both. We may also suppose by monotonicity that τ actually extends τ_s. In this case, we see if there exists a string $\tau \supset \tau_s$ such that $\Psi_e^\tau(x)$ converges.

If there is no such τ then $\Psi_e^B(x)$ will be undefined and since $A(x)$ will be defined because of being a total function, it does not matter what we do. In this case the requirement will be satisfied automatically and we may let σ_{s+1} be the smallest extension of σ_s defined on x. Since nothing has to be done on B, we let $\tau_{s+1} = \tau_s$.

If such τ do exist, then $\Psi_e^B(x)$ will be defined. Then we must define τ_{s+1} in such a way that B extends τ so that $\Psi_e^B(x) = \Psi_e^\tau(x)$ by monotonicity. It suffices if we let $\tau_{s+1} = \tau$, where τ is the first such we found. However, we have to be careful with A. Since $\Psi_e^B(x)$ is now defined, we need to make sure that it is different from $A(x)$. So, we let σ_{s+1} be the smallest extension of σ_s such that $\sigma_{s+1}(x) = 1 - \Psi_e^B(x)$.

If $s+1 = 2e+1$, then we just need to interchange the roles of A and B, the construction is the same.

Now A and B are computable in $\emptyset'$ since we use an oracle for $\emptyset'$ in constructing A and B. The only non-recursive step in the construction is where we ask, given x and σ, if there exists some σ' extending σ such that $\Psi_e^{\sigma'}(x)$ is defined. We consider all such σ' extending σ and we compute $\Psi_e^{\sigma'}(x)$ one step at a time in a dovetailing fashion. Hence, the construction is recursive in $\emptyset'$. □

Corollary 4. $\mathbf{D}$ is not linearly ordered.

The theorem can be extended to show that there is a set of $2^{\aleph_0}$ pairwise incomparable Turing degrees. One can also show that using a slightly different method that for every nonzero degree $\mathbf{a}$ there is a degree $\mathbf{b}$ incomparable with $\mathbf{a}$.

Exercise 7. Show that there exists a countably infinite sequence of degrees below $\mathbf{0}'$ each of which is incomparable with another.

Exercise 8. Prove the relativized version of Theorem 41. That is, show that for any degree $\mathbf{c}$, there are degrees $\mathbf{a}$ and $\mathbf{b}$ such that $\mathbf{c} \leq \mathbf{a}, \mathbf{b}$ and $\mathbf{a}, \mathbf{b} \leq \mathbf{c}'$ and that $\mathbf{a}$ and $\mathbf{b}$ are incomparable. (Hint: Use joins. For complete solution, see Soare's book, p. 133)

Exercise 9. A tree is a downward closed set of strings. A weak form of König's lemma asserts that every infinite binary tree has an infinite path. Show that this assertion is true. That is, given such a tree T, construct an infinite path on T using finite extensions.

Minimal pairs using Ψ_e-splittings

A typical problem is to get a degree incomparable with a given one. In the previous result, we built two sets simultaneously. Now we want to build a set incomparable to a given set. First we give the following notion.

Definition 35. Two strings σ_1 and σ_2 are Ψ_e*-splitting* if $\Psi_e^{\sigma_1}(x) \downarrow \neq \Psi_e^{\sigma_2}(x) \downarrow$ for some x. In this case, we say that σ_1 and σ_2 Ψ_e*-split* on x.

Now we show that there exists a pair of degrees with greatest lower bound.

Definition 36. Two degrees $\mathbf{a}$ and $\mathbf{b}$ form a *minimal pair* if they are non-zero and their greatest lower bound is $\mathbf{0}$, i.e.,

$$\forall \mathbf{c}(\mathbf{c} \leq \mathbf{a} \wedge \mathbf{c} \leq \mathbf{b} \Rightarrow \mathbf{c} = \mathbf{0}).$$

Theorem 42. There exists a minimal pair of degrees. In fact, each non-zero degree is a part of a minimal pair.

Proof. Let B be a non-recursive set. We want to define a set $A = \bigcup_s \sigma_s$ such that

$$\begin{array}{lcl} R_{2e} & : & A \neq \Psi_e \\ R_{2\langle e,i\rangle+1} & : & C = \Psi_e^A = \Psi_i^B \Rightarrow C \text{ is recursive} \end{array}$$

The first requirement ensures that A is non-recursive, while the second requirement ensures that any set recursive in both A and B is recursive. The first requirement is satisfed in the same way as we did in the previous theorem, i.e., we diagonalize against every recursive function.

The second requirement is satisfied by the splitting method. The idea to satisfy $R_{2\langle e,i\rangle+1}$ is the following. First we try to make the requirement vacuously true by having convergent computations such that

$$\Psi_e^A(x) \neq \Psi_i^B(x).$$

This can be done at a given stage by looking for Ψ_e-splitting extensions of σ_s. If they do exist, say τ_1 and τ_2, we choose the one that produces a disagreement with Ψ_i^B on the x for which the two strings Ψ_e-split. That is, we let $\sigma_{s+1} = \tau_k$, for the τ_k that disagrees, so that

$$B(x) \neq \Psi_e^{\tau_k}(x).$$

In this case the requirement is satisfied vacuously. If such strings do not exist, then let $\sigma_{s+1} = \sigma_s$. We still claim that Ψ_e^A is recursive if it is total. Suppose Ψ_e^A is total. Then by the use principle, given any x, there must exist some $\tau \subseteq A$ such that $\Psi_e^\tau(x) = \Psi_e^A(x)$. By the same principle, we can assume that τ extends σ_s. But since there are no Ψ_e-splitting extending σ_s, it must be the case that, as long as $\Psi_e^\tau(x)$ converges, the value is unique and equal to $\Psi_e^A(x)$. This suggests a recursive method to compute $\Psi_e^A(x)$: given x, dovetail the possible computations $\Psi_e^\tau(x)$, for all strings $\tau \subseteq \sigma_s$. The first converging one gives the right value of $\Psi_e^A(x)$. □

From a topological point of view, the difference between the previous proof, the pure finite extension method, and the splitting method is that the splitting method relies on something more than just simple continuity, since we use the fact that when a functional is constant on an open set, i.e., there is no splitting above a given string, then its value can be computed recursively.

Questions.

1. If Ψ^A is partial, does it make sense to write $\Psi^A = C$? Why/why not?

2. In Theorem 42, what step of the proof determines the the upper bound for the degree of A?

Coinfinite extension method

We said that $(\mathbf{D}, \leq)$ is not a (full) lattice for that it does not define a lower semi-lattice. $(\mathbf{D}, \leq)$ is a strict upper semi-lattice. That is, the least upper bound of any two degrees $\mathbf{a}$ and $\mathbf{b}$ always exists and it is defined as the join of $\mathbf{a}$ and $\mathbf{b}$. However, the greatest lower bound may not always exist. In order to show this we need to consider a phenomenon of 'exact pairs' for ideals of partial orders.

Definition 37. Let $(P, \leq)$ be a partial order and let $a_0 \leq a_1 \leq a_2 \leq \ldots$ be an increasing sequence of elements of P. A pair of elements (b, c) from P is said to be an *exact pair* for this sequence if

(i) $a_n \leq b, c$ for all n.

(ii) If $d \leq b, c$, then $d \leq a_n$ for some n.

Theorem 43. If $C_1 \leq_T C_2 \leq_T \ldots$ is an increasing sequence, then there are A and B such that (A, B) is an exact pair for this sequence.

Proof. Given a sequence $C_0 \leq_T C_1 \leq_T \ldots$ ascending in Turing degree, we define sets A and B such that

(i) $C_n \leq_T A, B$.

(ii) If $C \leq_T A, B$, then $C \leq_T C_n$ for some n.

To ensure these we need to satisfy the following requirements:

$$R_n : C_n \leq_T A, B \qquad\qquad N_{e,i} : \Phi_e^A = \Phi_i^B = C \Longrightarrow \exists n (C \leq_T C_n).$$

A finite extension argument will not be sufficient here. We will need what is called a *coinfinite extension* argument. In a coinfinite extension argument, at each stage of the construction we define A and B on infinitely many arguments but at the end of each stage we also leave them undefined on an infinite number of arguments.

We build A and B, which are divided into columns, by approximations α_s and β_s. But instead of taking α_s and β_s as finite strings, we consider them as matrices. We consider each α_s as the partial function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ which specifies A on arguments we have already decided by stage s. In each matrix, finitely many columns are entirely determined and there is finitely much additional information. Suppose at stage s we work for R_n. Choose the first column in each of α_s, β_s which has no specifications as yet. Let α_{s+1} (similarly β_{s+1}) be the result of putting C_n into that column of α_s (similarly β_s) and leaving the rest of the approximation unchanged. This action is computable in C_n. Otherwise, suppose at stage s we work to satisfy $N_{e,i}$. Ask if there exists some x, some $\alpha \supseteq \alpha_s$ and $\beta \supseteq \beta_s$ such that $\Phi_e^\alpha(x) \downarrow = \Phi_i^\beta(x) \downarrow$ with the domains of α and β being only finitely larger than those of α_s and β_s, respectively. If such extensions exist, let $(\alpha_{s+1}, \beta_{s+1})$ be the least such pair of extensions. If no such extensions exist, do nothing.

A and B meet the condition that $C_n \leq_T A, B$ for all n, because all the R_n requirements are satisfied. Consider the stage s at which we deal with requirement $N_{e,i}$. We may assume that $\Phi_e^A = \Phi_i^B = C$ as otherwise the requirement is automatically satisfied. We want to prove $C \leq_T C_n$ for some n. Indeed let n be the largest m such that we have coded C_m into A and B by stage s. To Compute $C(x)$, we use an oracle for $D = \bigoplus_{k=0}^{s-1} C_k$, and so is of degree in the ascending sequence (since the degree of D is bounded by the degree of C_{k+1}). Certainly D can decide which arguments α_s and β_s are defined on, and can compute their values on all such arguments. We find any finite extension α of α_s such that $\Phi_e^\alpha(x) \downarrow$ (There is one since $A \supseteq \alpha_s$ and $\Phi_e^A(x) \downarrow$ if it is total). We claim that $\Phi_e^\alpha(x) = \Phi_e^A(x) = C(x)$. Suppose otherwise. Then let β be a finite extention of β_s with $\beta \subseteq B$ such that $\Phi_i^\beta(x) \downarrow$. Since $\Phi_e^A = \Phi_i^B$, it must be the case that $\Phi_e^\alpha(x) \neq \Phi_i^\beta(x)$ which is impossible by hypothesis. □

Corollary 5. D is not lattice, i.e., it is a strict upper semi-lattice.

Proof. Let $\langle C_i \rangle$ be strictly increasing in Turing degree (for example we may let $\mathbf{c}_{i+1} = \mathbf{c}_i'$). Then, by Theorem 43, an exact pair (A, B) exists for this sequence. If there were some C whose degree is the greatest lower bound of those A and B, then $C \leq_T A, B$ but also $C \leq_T C_n$ for some n. Then, $C <_T C_{n+1} \leq_T A, B$. Hence, C is not the greatest lower bound of A and B. A contradiction. □

6.2 Simple sets and permitting method

Two basic methods for controlling the degrees of sets we construct are *coding* and *permitting*. Coding is a way of ensuring that a set A we build has degree at least that of a given set B. As the name implies, it consists of encoding the bits of B into A in a recoverable way. One simple way to to do this is to build A so that $A = B \oplus C$ for some C. Permitting is used to define a set A such that $A \leq_T B$ for a given set B. The simplest version of the permitting method is used when we construct an r.e. set A such that $A \leq_T B$ for a given r.e. set B, where we only define x to be in A_s if some element $y \leq x$ appears in B_s. In this case, we say that B *permits* x to enter A at stage s. To see why this guarantees that $A \leq_T B$, observe the following theorem.

Theorem 44. If A and C are r.e. sets such that $\{A_s\}_{s\in\mathbb{N}}$ and $\{C_s\}_{s\in\mathbb{N}}$ are recursive enumerations of them, and if for every x

$$x \in A_{s+1} - A_s \Longrightarrow (\exists y \leq x)(y \in C_{s+1} - C_s),$$

then $A \leq_T C$.

Proof. To see if $x \in A$, look at the stages in which some $y \leq x$ is generated in C: recursively in C we may determine if $y \in C$, and if so we just have to generate C until y appears in it. Then $x \in A$ if and only if x is generated at one of these stages. □

As an example we will show how to apply the permitting method to construct what is known as a 'simple' set below a given non-recursive r.e. set. Recall that every infinite recursively enumerable set contains an infinite recursive subset. This was proved by Post in Theorem 13. Some infinite sets though do not contain any r.e. subset. An infinite set is called *immune* if it does not contain any r.e. subset.

Earlier we also talked about creative sets (see Definition 16), i.e., sets whose complement is productive. Creative sets are those which are computably isomorphic to the halting set K. We now give other examples to incomputable sets. We know that, by definition, the complement of every creative set contains an infinite r.e. subset. It would be quite different from K if we could find an incomputable r.e. set whose complement does not have that property.

Definition 38. An r.e. set S is called *simple* if

(i) $\overline{S}$ is infinite

(ii) For every $e \in \mathbb{N}$, if W_e is infinite then $W_e \cap S \neq \emptyset$.

In other words, a simple set is a recursively enumerable set whose complement is immune. First we observe that simple sets are incomputable. The item (ii) in the definition ensures that S is non-recursive.

Proposition 12. If S is simple, then S is not computable.

Proof. Suppose, for a contradiction, that S is simple and computable. Then $\overline{S}$ is infinite, by definition, and computable. This means that $\overline{S}$ is an infinite r.e. set and so it must be equal to some infinite W_e. But this contradicts (ii) in the definition. □

Let us now construct a simple set as they do not naturally occur. We construct them using a primitive version of what we will call later a *priority argument*.

Theorem 45. There exists a simple set.

Proof. (Version 1—Due to Post, 1944) Let $A = \text{ran}(\psi)$ where

$$\psi(i) = \mu n > 2i \text{ such that } n \text{ is enumerated into } W_i.$$

Since ψ is partial recursive, A is r.e. If $x < 2i$ is in A, then $x = \psi(k)$ for some $k < i$. Hence, $|A \cap [0, 2i)| \leq i$, so $\overline{A}$ is infinite. Then, by definition, A is simple. □

Proof. (Version 2) We present the following proof in a different way to introduce some new terminology. Roughly the idea is that for each e we wait for some stage s at which some x is enumerated into $W_{e,s}$. We then enumerate x into A to make $A \cap W_e \neq \emptyset$. However, we have to be a little careful about which x we use if we are to have $\overline{A}$ infinite. Of course we can only use computable information in choosing x if we are to end up with a set A which *seems* to be recursively enumerable.

We effectively enumerate the members of A at each stage s in order to satisfy the *requirements*: For all $e \in \mathbb{N}$,

$\mathcal{P}_e$: If W_e is infinite, then $W_e \cap A \neq \emptyset$.
$\mathcal{N}_e$: $|A \cap \{0, 1, \ldots, 2e\}| \leq e$.

The $\mathcal{P}_e$ requirements ensure the condition (ii) of the definition of simple set, whereas the $\mathcal{N}_e$ requirements ensure that $\overline{A}$ is infinite. For part (i) of the simplicity definition, notice that if $\mathcal{N}_e$ holds then $|\overline{A} \cap \{0, 1, \ldots, 2e\}| > e$. So if $\mathcal{N}_e$ holds for infinitely many e,

then $\overline{A}$ must be infinite. $\mathcal{P}_e$ is just part (ii) of the definition. So if these requirements are satisfied, then A is simple.

Formal construction of A:

1. For each as yet *unsatisfied* $\mathcal{P}_e$, wait for a stage s at which there is a member $x \in W_{e,s}$ with $x > 2e$.

2. If such x appears, enumerate x into A, at which stage $\mathcal{P}_e$ becomes *satisfied*.

Note that if no such x appears, then W_e is finite and the requirement is satisfied vacaously.

Let us now give the verification. In this case, the verification consists of the observation that A is r.e., by the effectiveness of the procedure for enumerating the members of A, and proofs of two simple lemmas.

Lemma 3. $\mathcal{P}_e$ is satisfied for each $e \in \mathbb{N}$.

Proof. Assume that W_e is infinite. Let s be the least stage at which we get some $x \in W_{e,s}$ with $x > 2e$. By the construction at stage s, $\mathcal{P}_e$ is not already satisfied, which means that it becomes so by step (2) of the construction at stage s via $x \in W_e \cap A$. □

Lemma 4. $\mathcal{N}_e$ is satisfied for each $e \in \mathbb{N}$.

Proof. Since we can only enumerate a number x into A with $x > 2e$ on behalf of some W_i with $i < e$, and for each i at most on such x is enumerated, the lemma follows immediately and this proves the theorem. □

Exercise 10. Observe that not all non-computable r.e. sets are simple, since given any non-computable r.e. set A, the set $\{2n : n \in A\}$ is also r.e. and non-computable, yet it is not simple. Argue why this is the case.