A Quest for Intuition: Understanding Computability and Complexity

Mohammad A. Mahmoud

December 3, 2024

Contents

Chapter 1

Introduction

1.1 Motivation

Computability Theory and Complexity Theory are foundational pillars. The first is concerned mainly with what computers cannot do, and it also delineates the frontier between what computers can and cannot do. It offers insight into the capabilities and limitations of computation. The second is concerned with the resources required for computation. Computability Theory has been mostly an academic curiosity, whereas Complexity Theory is essential for the design and analysis of algorithms, the optimization of processes, and the development of new technologies. That said, Complexity Theory is built on the formal grounds laid by Computability Theory.

This book is to be considered an expository work that introduces the basics of a large area of research. It aims to succinctly and intuitively deliver a bouquet of topics that normally appear as a part of a much heavier textbook, or can be found scattered online.

1.2 Preliminaries

We need to recall some basic concepts and notation that will be used throughout this book.

We will use $\mathbb{N}$ to denote the set of natural numbers including 0, that is $\mathbb{N} = \{0, 1, \ldots\}$.

For a number k, we will use $\mathbb{N}^{<k}$ to denote the set of natural numbers less than k, that is

$$\mathbb{N}^{<k} = \{n \in \mathbb{N} \colon n < k\}.$$

Similarly, we will use $\mathbb{N}^{>k}$, $\mathbb{N}^{\leq k}$, and $\mathbb{N}^{\geq k}$.

As commonly used, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ will denote the integers, the rationals, the reals, and the complex numbers, respectively.

For a set X, we use $\mathcal{P}(X)$ to denote the set of all subsets of X. The set $\mathcal{P}(X)$ is known as the *power set* of X.

When we write $f \colon D \to C$ we mean that f is a function with *domain* D and *co-domain* C. Recall that the co-domain is not the same thing as the *range*. The range, sometimes called the *image*, is the set of all outputs of the function, more clearly: $range(f) = \{f(d) \colon d \in D\}$.

Clearly, the range is always a subset of the co-domain. If it happens that the range = the co-domain, we say that the function is *surjective* or *onto*.

Mentioning the words "surjective" and "onto", you are probably remembering of the words "injective" or "one-to-one" (aslo often written "1-1"). We say a function is *injective* (or *one-to-one*) if no two different elements in the domain are mapped to the same image (output). More clearly, the function f is injective if [1]

$$(\forall x, y \in D)[x \neq y \Longrightarrow f(x) \neq f(y)].$$

or equivalently (why?),

$$(\forall x, y \in D)[f(x) = f(y) \Longrightarrow x = y].$$

If a function is both injective and surjective, we call it *bi-jective*.

Exercise 1 Suppose A and B are two sets such that there exists a bijection from A to B. In other words, there exists some bijective function $g\colon A \to B$. Prove that: There exists a bijective function from B to A.

Exercise 2 Suppose A and B are two sets such that there is no injection with domain B and co-domain A. Prove that there is no surjective function from A to B.

For a set A, we will write $|A|$ to denote the *cardinality* of A, which informally means the number of elements in A. This is informal because the number of elements in a set does not make clear sense when the set is infinite. If you do not recall the formal meaning of cardinality, no problem, the informal meaning will be good enough to understand this book.

For sets A, B, if we write $|A| \leq |B|$, then we means that there exists an injection from A to B. Clearly, if the inequality is strict, then it means in addition that there is no injection from B to A (or, equivalently, that there is no bijection).

Exercise 3 Prove that: For every set X, $|X| < |\mathcal{P}(X)|$.

A set A is considered *countable* if $|A| \leq |\mathbb{N}|$, and *uncountable* otherwise.

Exercise 4 Prove that $\mathcal{P}(\mathbb{N})$ is uncountable.

[1] Question: "if" or should it be "iff" ?

Exercise 5 Prove that the countable union of countable sets is a countable set.

Back to when we wrote $f\colon D \to C$ to mean that f is a function with domain D and co-domain C. Recall that, algebraically, f is a set, namely: $f = \{(x, f(x))\colon x \in D\}$, which is a subset of the *cartesian product* $D \times C$.

Some authors prefer to call the set $\{(x, f(x))\colon x \in D\}$ the *graph of* f, and would consider the function itself a combination the domain, co-domain, and the formula that gives the outputs from the inputs. This approach is usually a hides a bit of informality.

Remark 1 If $f \subseteq A \times B$, and f is a mapping (i.e. $(\forall a \in A, b_1, b_2 \in B)[((a, b_1) \in f \wedge (a, b_2) \in f) \implies b_1 = b_2]$), then f is not necessarily a function from A to B. However, it is a function from D to B where $D = \{a \in A\colon (\exists b \in b)[(a, b) \in f]\}$. This is how the domain can be defined from the set definition of a function.

Tip 1 When defining functions, make it a habit to explicitly mention the domain and co-domain. For example, it is not proper to say "Let f be the function $f(x) = x^2$". A proper way would be: "Let f be the function with domain $\mathbb{R}$ and co-domain $\mathbb{R}$ given by $f(x) = x^2$", or "Consider $f\colon \mathbb{R} \to \mathbb{R}$ given by $f(x) = x^2$". As you read this book, pay attention to how we write things.

Recall what we mean by a (binary) relation between two sets. For sets D and C, any subset of $D \times C$ can be called a *relation between* D and C (or better say from D to C because the order obviously matters). As you can see, a every function from D to C is a relation from D to C. In fact, you should recall the following:

$$f \text{ is a function from } D \text{ to } C \text{ if}$$

$$f \subseteq D \times C,$$

and

$$(\forall x \in D)(\forall y_1, y_2 \in C)\left[((x, y_1) \in f \wedge (x, y_2) \in f) \implies y_1 = y_2\right].$$

Exercise 6 Show that the definition of injectivity does not change if we replace $\implies$ by $\iff$.

A relation from A to A is called a relation *on* A.

Example 1

The famous $<$ is a binary relation on $\mathbb{N}$. Indeed,

$$< = \{(0,1),(1,2),(0,2),(2,3),(1,3),(0,3),\ldots\}^{2}.$$

However, $<$ is not a function (or a mapping) because $(0,1)$, $(0,2)$ are both elements in $<$.

A *sequence* is a function with domain $\mathbb{N}$. A *string*, or a *finite sequence*, or a *finite tuple*, all mean a function with domain $\mathbb{N}^{<k}$ for some natural number k (which is the *length* of the string).

Remark 2

Depending on the context, the empty set $\emptyset$ may be called: "empty function", "empty relation", "empty string", etc. This makes sense because all functions, relations, etc. are all sets.

Soon you will be seeing programs as strings. Indeed, in any syntax, a program is a finite sequence of symbols. This is why, at some point in this book, we will talk about *empty programs* and it will make sense.

We talked about binary relations, which are a special kind of finitary relations. Generally, for any natural n, R is an *n-ary relation* on sets $A_1,\ldots,A_n$ if R is a subset of the cartesian product $A_1 \times \ldots \times A_n$. If all the A_i's are the same set A, we denote the cartesian product by A^n, and say R is an n-ary relation on A.

Remark 3

0-ary relations are known as *nullary*, 1-ary as *unary*, 2-ary as *binary*, and 3-ary as *ternary*.

A unary relation on a set A is simply a subset of A.

A relation can by seen as a logical function (predicate). For example, we have seen $<$ as a subset of $\mathbb{N} \times \mathbb{N}$, but we can also see it as a function:

$$<: \mathbb{N} \times \mathbb{N} \to \{\ \text{TRUE}\ ,\ \text{FALSE}\ \},$$

where

$$<(x_1,x_2) = \ \text{TRUE} \iff x_1 < x_2.$$

One may say that we call sets "relations" when we want to think about them as predicates (Boolean functions).

Vacuous Truth

Many statements in mathematics are *universal statements*, which are usually some version of the following:

$$(\forall x \in \mathcal{U})P(x),$$

where $\mathcal{U}$ is some universe about which the statement is made (a domain of interpretation), and $P(x)$ is some predicate.

It is helpful to understand such a statement as $(\forall x)[x \in \mathcal{U} \implies P(x)]$.

Often it happens that a universal statement is *vacuously true*, what does that mean?

For any universal statement $(\forall x \in \mathcal{U})P(x)$, if $\mathcal{U}$ is empty, then the statement is true (or we say, vacuously true). The reason is, as we said earlier, the statement can be rephrased as $(\forall x)[x \in \mathcal{U} \implies P(x)]$, which is equivalent to $(\forall x)[\text{ FALSE } \implies P(x)]$. As we know, the implication FALSE $\implies P(x)$ is true regardless of the truth value of $P(x)$.

Practice Problems

1. Let $f\colon A \to B$ and $g\colon B' \to C$. If $range(f) \subseteq B'$, then we can define a function $h\colon A \to C$ such that $(\forall x \in A)[h(x) = g(f(x))]$. The function h is known as the *composition* of g and f. It is common to write h as $g \circ f$.
   (a) Prove that: if f, g above are injective, then $g \circ f$ is also injective.
   (b) Find an example where f, g are surjective, but $g \circ f$ is not.
2. Let A be a non-empty set. Is there an empty function with co-domain A? Is it unique?
3. True or False:
   (a) An empty function is always injective.
   (b) An empty function can be surjective.
   (c) There exists a bijective empty function.
4. **Cantor's Pairing**[3] Consider the following function
   $$\pi\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N},$$
   given by
   $$\pi(x_1, x_2) = \frac{1}{2}(x_1 + x_2)(x_1 + x_2 + 1) + x_2.$$
   Prove that π is bijective.
5. Prove that: For any natural number $m > 0$, we have $|\mathbb{N}^m| = |\mathbb{N}|$. Do not use the Shröder - Bernstein Theorem.
6. What is $|\mathbb{N}^0|$?
7. Find functions $\pi_1, \pi_2\colon \mathbb{N} \to \mathbb{N}$ such that $\pi(\pi_1(z), \pi_2(z)) = z$.
8. Recall that a polynomial on $\mathbb{Z}$ in one variable x is an expression of the form $\sum_{i=0}^{m} a_i x^i$ where, for all i, $a_i \in \mathbb{Z}$. The a_i's are known as the *coefficients*.
   The same concept can be extended to include more than one variable. A polynomial on [4] $\mathbb{Z}$ in k variables $x_1, \ldots, x_k$ is an expression of the form
   $$\sum_{i_1, i_2, \ldots, i_r \in \{0, \ldots, m\}} a_{i_1, i_2, \ldots, i_r} x_1^{i_1} \ldots x_k^{i_r}$$

[3] Any bijection from $\mathbb{N}^2$ to $\mathbb{N}$ is a pairing function. Cantor's pairing is one example of a computable pairing function. Soon you will understand what computable means. It is common to denote pairing as $\langle\ .\ ,\ .\ \rangle$ in the sense that $\langle x, y\rangle = p(x, y)$ where p is whichever pairing function being used.

[4] Generally, we can have polynomials on any *ring* $\mathbb{K}$. A ring is an algebraic structure that allows two operations with certain properties, which are usually called addition and multiplication. Examples of rings: $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. For more on rings and general (algebraic) structures, see the appendix.

where all the coefficients $a_{i_1,i_2,\ldots,i_r}$ are integers.

A *Diophantine equation* is a polynomial equation on $\mathbb{Z}$ in two or more variables. That is, a Diophantine equation in variables $x_1, x_2, \ldots, x_r$ is an equation that can be put in the following form:

$$\sum_{i_1,i_2,\ldots,i_r \in \{0,\ldots,m\}} a_{i_1,i_2,\ldots,i_r} x_1^{i_1} \ldots x_k^{i_r} = 0,$$

where the coefficients are all integers.

Example: Let $f(x_1, x_2, y_1, y_2) = x_1^2 - 2y_1^3 + y_1 y_2^3$. Then, $f(x_1, x_2, y_1, y_2) = 12$ is a Diophantine equation in the variables x_1, x_2, y_1, y_2 (or in the variables x_1, y_1, y_2).

To see how the general form works for the example (assuming we want to see it as an equation in x_1, x_2, y_1, y_2), complete the following values $m = 3$, $a_{2,0,0,0} = 1$, $a_{0,0,3,0} = -2$, $a_{0,0,1,3} =??$, $a_{0,0,0,0} =??$, $a_{1,1,1,1} =??$.

9. Let j be a natural number, and let A be a subset of $\mathbb{N}^j$. We say that A is a *Diophantine subset* if there exists a polynomial equation in at least j variables, say, $P(x_1, \ldots, x_j, y_1, \ldots, y_k) = 0$ such that

   For all $x_1, \ldots, x_j \in \mathbb{N}$,

   $$(x_1, \ldots, x_j) \in A \Longleftrightarrow (\exists y_1, \ldots, y_k \in \mathbb{N})[P(x_1, \ldots, x_j, y_1, \ldots, y_k) = 0].$$

   (a) Prove that the set of even numbers is a Diophantine subset of $\mathbb{N}$.

   (b) Prove that the set of Pythagorean triples is a Diophantine subset of $\mathbb{N}^3$.

   (c) prove that the relation $<$ is a Diophantine subset of $\mathbb{N}^2$.

   (d) Prove that the set of prime numbers is the complement of a Diophantine subset of $\mathbb{N}$.

   (e) Prove that the union (intersection) of two Diophantine subsets of $\mathbb{N}^j$ is also a Diophantine subset.

Part I

Theory of Computability

The motivation to study Computability is to understand ourselves better; how us humans compute. Computability theory helps us discern which problems are solvable by algorithms and which are inherently unsolvable, providing a clear demarcation between the possible and the impossible in computation.

Chapter 2

Building a machine

2.1 Introduction

This book is an attempt to introduce the common concepts of the theory of computation as a natural answer to some of our most basic questions about mathematics, the kind of questions which we needed to answer to build calculators and computers, or to just understand ourselves better.

Throughout this textbook, you will be asked some very natural simple questions, but they will progressively make you delve deep into analyzing how you think.

If you want to build a machine to do something for you, you need to break your thought *process* down to the most *primitive* steps. You need to understand yourself better.

Before you start reading the first chapter, think of a very basic procedure/machine [1] by which you can perform addition. Think of small numbers, it does not need to handle values > 10.

More questions to think about:

Is there any process that is more basic than addition?

How do you understand, say, the quantity: 5 ? What is the message you understand from seeing the character '5' ? What is a good visual representation of $2 + 3$ as a quantity?

[1] You will later see why I put the two words together. At some point we will consider procedures and machines as the same thing

At the end of the introduction you were asked to think of a basic procedure/machine by which you can perform addition.

You probably thought of counting fingers or apples. You also might have thought of a fancy abacus.

Do you notice something here? Any of the examples (fingers, apples, abacus) use **counting**. These examples also rely on (which is too primitive to notice) our ability to understand numbers through the unary representation. When you see three apples, you understand something.

What happens when you see "3" ?

To summarize and generalize, any (physical) machine for computing requires:

1. Objects: to give a physical presence to numbers
2. Storage: to work with the objects
3. An input/output convention: to communicate with humans.

Obviously, the three requirements above are not everything. None of the requirements above describes how the machine should perform any actions.

The procedures/machines we mentioned (fingers, apples, abacus) perform addition by "bringing together" the objects. To explain this point, consider the following scenario:

You asked a small child who just learnt about addition: what is 2 plus 5 ? You gave the child a box with enough many apples and asked them to use the apples to assist them in finding the answer. The child learnt to do the following:

1. Put 2 apples together somewhere on a table
2. Put 5 apples together on the same table
3. Count all the apples on the table.

The next day, the child learnt about subtraction. Now you ask them: What is 7 minus 3 ? Given the same table and box of apples, the child proceeded to do the following:

1. Put 7 apples together somewhere on the table
2. Removed 3 apples from the table and returned them into the box (out of view)
3. Counted all the apples on the table.

What is the point from these scenarios?

As you can probably see, the procedures relied on some primitive abilities such as add, remove, move things around (closer or further from each other). There is a conceptual machine that does these exact tasks, it is called the Turing machine.

Turing's Machine

In 1936, Alan Turing described a machine in the spirit of the ideas we have just discussed. The machine has (see Figure 2.1.1):

1. memory cells, which can be imagined contiguous in one line on a tape;
2. a reading/writing head.

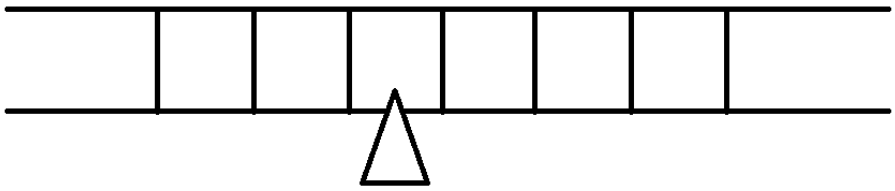

Figure 2.1.1: Head and Tape

In addition, the same way we had units (e.g. apples) to use for counting, we have characters (e.g., '0', '1', 'a', ...) which we can put in the memory cells. We assume that each cell can contain at most one character. The simplest situation assumes we have a single character (say, '1' only) which we can use repeatedly to represent quantities. For example, we can represent 3 on the tape as:

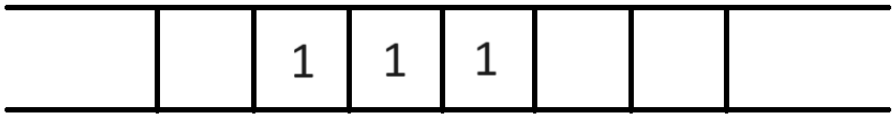

Figure 2.1.2: The value 3 on the tape

In the same way the child had enough many apples and enough space on the table, conceptually, we can assume the tape is infinite in both directions and that there are infinitely many of the character '1' to be able to put any finite number we want on the tape. Another way to see it, the head can write 1 on any cell at any time.

Since the child was able to move the apples around the table, we allow the head to delete and move as well.

It is reasonable to assume that the following is the simplest set of actions the head can do:

1. write '1'

2. delete '1' (empty a cell)
3. move one cell to the Left
4. move one cell to the Right

Using these actions, the head can access and change any cell on the tape. Now think of this machine as a simple abstract computer, or as a fancy abstract abacus.

Suppose now you want to compute a function $f\colon D \to \mathbb{N}$ on this kind of machine (where $D \subseteq \mathbb{N}$). First thing, you will write the input (or inputs) on the tape. Obviously, you will need to fix a way (convention) according to which the inputs will be written. Secondly, you will need to specify actions for the head to do. More clearly, you will need to provide a set a of instructions (a program) to tell the head what to do, say, if it is at a cell that contains '1', or if it is at an empty cell. Finally, you will need to specify a convention for interpreting the output.

A common input/output convention is the following:

Input Convention: If the intention is to give the machine inputs $(x_1, \ldots, x_k)$, populate the tape (left to right) starting at the cell where the head is located with: $x_1 + 1$ many 1's, followed by an empty cell, followed by $x_2 + 1$ many 1's, followed by an empty cell, and so until we include the last input. For example, (2,0,1) will look like:

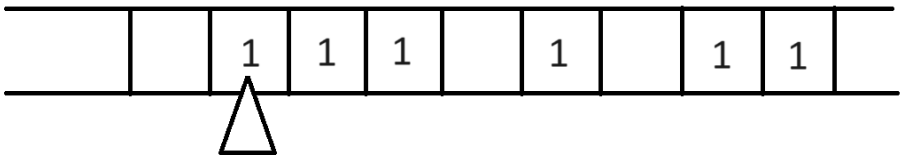

Figure 2.1.3: (2, 0, 1) on the tape

Probably you can see why we are using $+1$ instead of the number itself, to reasonably represent the value 0.

Output Convention: The output is the number of 1's left on the tape after the program stops, i.e., when none of the instructions is anymore applicable. It does not matter if they are contiguous, and it does not matter where the head will be.

According to these conventions, the following is a set of instructions (a Turing program) that computes the function $g\colon \mathbb{N} \to \mathbb{N}$ such that $g(x) = x + 3$.

Repeat the following until no instructions are applicable or a halt is reached:

If the head is at a nonempty cell:

move to the right

Else:

write 1

move to the right

write 1

halt (exit the program)

You should be able to see that following these instructions makes the machine write two extra 1's on the tape. For example, if the machine starts at the configuration where three 1's on the tape and the head is at the leftmost 1 (input =2), then it will finish with a configuration where there is five 1's on the tape.

Here is a visualization of the configurations reached by the machine while following the instructions:

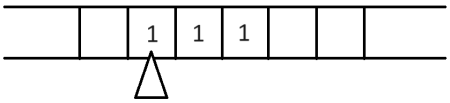

Figure 2.1.4: 1st Configuration

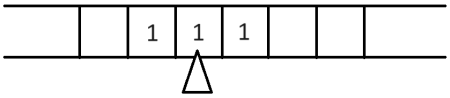

Figure 2.1.5: 2nd Configuration

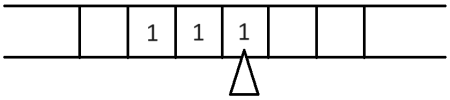

Figure 2.1.6: 3rd Configuration

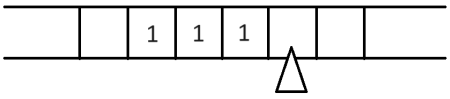

Figure 2.1.7: 4th Configuration

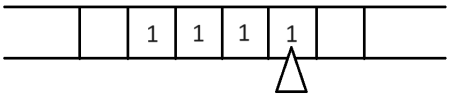

Figure 2.1.8: 5th Configuration

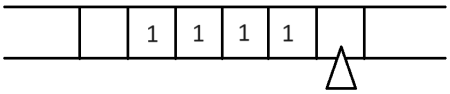

Figure 2.1.9: 6th Configuration

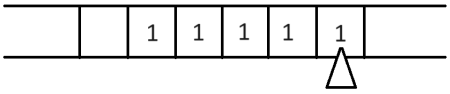

Figure 2.1.10: 7th Configuration

So far, it is not clear how the head follows instructions. The instructions were written in a way us humans can understand, as if the head is a human following the instructions.

It is not difficult to imagine a real machine that can execute the first two lines of the instructions (perhaps the head has a sensor and 1 means there is an electric signal at the cell). But what about the Else part? The instructions after Else form a subroutine, should we save that subroutine in the head? what if there is a sub-subroutine?

We introduce the concept of *state*. If the head is a human, they will have one state of mind if they find 1, and another state of mind if they find an empty cell. Based on the state of mind, the head (the human) knows which actions to take. In the example we have just discussed, there are two states for the mind of the head. One state makes the head move to the right, while the other makes the head write 1, move to the right, then write 1 again.

Giving the machine instructions can now be understood as putting the head in particular states. The instructions given before can be rewritten using three states, let us call them state0, state1, and state2 (we assume the machine starts with the head in state0):

If the head is at state0 at a nonempty cell:

move to the right

If the head is at state0 at an empty cell:

change to state1

write 1

move to the right

If the head is at state1 at an empty cell:

change to state2

write 1

move to the right

Or, to make the routines of all the states look the same, we can write:

If the head is at state0 at a nonempty cell:

change to state0 *#which keeps the state unchanged*

write 1 *#overwrite the already existent '1'*

move to the right

If the head is at state0 at an empty cell:

change to state1

write 1

move to the right

If the head is at state1 at an empty cell:

change to state2

write 1

move to the right

If you notice there is no instruction for when the head has state2 at a nonempty cell. This means that if the machine reaches such a situation it will halt.

The programs above can be written in a more economic way by using symbols as follows:

$$q_0\ 1\ q_0\ 1\ R$$
$$q_0\ b\ q_1\ 1\ R$$
$$q_1\ b\ q_2\ 1\ R$$

To explain, q_0, q_1, q_2 are states, b means blank (empty cell), and R means to move to the right. A quintuple such as $q_0\ b\ q_1\ 1\ R$ means: If the head is in the state q_0 at an empty cell, then the head will change to be in state q_1, write 1, and move to the right. As you can see, any line in our Turing program takes the form

$$q_i\ m\ q_j\ n\ D$$

where $m, n \in \{1, b\}$, $i, j \in \mathbb{N}$, and $D \in \{L, R\}$. Obviously by 'L' we mean "move to the left".

Following up on the previous example, generally speaking, m, n come from a set called the *tape alphabet*, which can be any finite set of symbols. To summarize:

Definition 1 **(Turing machine - Informal)**

A Turing machine is a conceptual machine which is made from two parts:

1. a two-way infinite tape that is divided into cells
2. a head that can move along the tape

The head is programmed using states, and can do one of the following actions at a time:

1. read a symbol from a cell
2. write a symbol on a cell
3. delete a symbol from a cell
4. move to the left
5. move to the right
6. change its state

The symbols come from a finite set which we call the tape alphabet, and there is only finitely many possible states.

From a mathematical point of view, the definition above is considered "informal" for obvious reasons. For example, what does "tape" mean mathematically?

The following is an attempt inspired by Hopcroft and Ullman [**?**] to create a mathematical entity that captures the essence of what a Turing machine is, in a very general form. This formal general definition, even though it looks mysterious and cumbersome, it is convenient for describing what matters mathematically in a Turing machine.

Definition 2 **(Turing machine - Formal)**

A Turing machine is a 7-tuple $M = (Q, q_0, F, \Gamma, b, \Sigma, \delta)$ where

1. Q, Γ are finite nonempty sets,
2. $q_0 \in Q$ and $b \in \Gamma$,
3. $F \subseteq Q$ and $\Sigma \subseteq \Gamma \setminus \{b\}$,
4. δ is a partial function from $(Q \setminus F) \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$.

This completes the definition of the mathematical entity which we call a Turing machine. This mathematical entity (the 7-tuple) is interpreted as follows:

1. Q is the set of states.
2. Γ is the tape alphabet.
3. q_0 is the initial state (the default state of the head at the beginning of any use).
4. b (from "blank") means empty cell.
5. F (from "final") are halting states[2]. That is, if the head reaches any of these states, the program stops.
6. Σ are the symbols allowed for writing inputs, which is a subset of Γ, the symbols that can be put on the tape during computation and to give the output.
7. δ defines the actions of the head. For example, if $\delta(q_0, 1) = (q_1, b, L)$, then this means that: if the head is in state q_0 at a cell that contains 1, then it will change to state q_1, delete the 1 from the cell, then move one cell to the left.

Remark 4 You can find different definitions in literature, but the resulting machine has the same computational power (this will make more sense after we study computable functions).

Remark 5 As you know, a function is a set of ordered pairs. The transition function δ of the previous definition is a subset of $((Q \setminus F) \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$. More clearly, the elements of δ look like $((q_i, m), (q_j, n, D))$ where $m, n \in \Gamma$, $q_i \in Q \setminus F$, $q_j \in Q$, and $D \in \{L, R\}$.

For example, the machine in Example **??** has $\delta = \{((q_0, 1), (q_0, 1, R)), ((q_0, b), (q_1, 1, R)), ((q_1, b), (q_2, 1, R))\}$. As you can see, δ is literally the Turing program.

Remark 6 It is common to write the transition function δ in a table. For example, the δ of Example **??** can be written as:

Insert Table

Example 1

According to Definition 2, a machine that fits for Example **??** can have $Q = \{q_0, q_1, q_2\}$ (or any finite superset of that), $\Gamma = \{1, b\}$, $\Sigma = \{1\}$, F can be any subset of Q (can even be the empty set), and as explained in the previous remark, $\delta = \{((q_0, 1), (q_0, 1, R)), ((q_0, b), (q_1, 1, R)), ((q_1, b), (q_2, 1, R))\}$.

Remark 7 According to Definition 2, and even according to Definition 1, a Turing machine comes with a program. In other words, a Turing machine does one particular task. This is a reason why it is common to identify a machine with its program (assuming all the parameters $Q, \Gamma, \ldots$ etc. are understood).

For example, we can say that the following is a Turing machine:

[2] You might be wondering why there is a need for more than one halting state.

$$q_0\ 1\ q_0\ 1\ R$$
$$q_0\ b\ q_1\ 1\ R$$
$$q_1\ b\ q_2\ 1\ R$$

even though it is not a 7-tuple.

In this book we introduce the following definition (which will be helpful later):

Definition 3 **(Turing Machine as a Turing Program)**

A Turing machine (or program) is a finite sequence (a string) on the form

$$\{q_{i(l)}\ m(l)\ q_{j(l)}\ n(l)\ D(l) \colon l \text{ natural } < k\}$$

where k is some natural number (the length of the program), i, j are functions from $\{0, \ldots, k-1\}$ to $\mathbb{N}$, and m, n are functions from $\{0, \ldots, k-1\}$ to $\{b, 1\}$, and D is a function from $\{0, \ldots, k-1\}$ to $\{L, R\}$.

If we separate the elements of the sequence in lines, the sequence will look like:

$$q_{i(0)}\ m(0)\ q_{j(0)}\ n(0)\ D(0)$$
$$\vdots$$
$$q_{i(k-1)}\ m(k-1)\ q_{j(k-1)}\ n(k-1)\ D(k-1)$$

Question 1 Can k in the above definition be 0? What would the corresponding transition function δ be?

I will add an appendix on the logical meaning of a sequence or a string

Strings vs Natural Numbers

According to Definition 2, any input or output is a (finite) string made of some tape alphabet symbols. Therefore, a Turing machine can be regarded as a function from a set of finite strings to another set of finite strings.

It is helpful to learn that every string on a finite alphabet can be encoded as a natural number. There are many ways to do so, including those implemented in modern computers. Here you will see two popular techniques that are good enough for mathematical purposes.

Pairing

Recall that a pairing function is a bijection from $\mathbb{N}^2$ onto $\mathbb{N}$, and it is common to write such a function $\langle\ .\ ,\ .\ \rangle$.

Consider now the following function

$$f_3 \colon \mathbb{N}^3 \to \mathbb{N}$$

such that

$$f_3(x_1, x_2, x_3) = \langle\langle x_1, x_2\rangle, x_3\rangle.$$

Similarly, for $n > 3$, we have

$$f_n \colon \mathbb{N}^n \to \mathbb{N}$$

such that

$$f_n(x_1, \ldots, x_n) = \langle f_{n-1}(x_1, \ldots, x_{n-1}), x_n\rangle.$$

Question 2 To define f_n, did we use primitive recursion? If not, can the definition be formalized using primitive recursion? [Move this from here, they have not learnt about PRIM yet]

Generally, we can assume that $(x_1, \ldots, x_n)$ is encoded as $\langle\langle \ldots \langle\langle x_1, x_2\rangle, x_3\rangle, \ldots, x_{n-1}\rangle, x_n\rangle$.

Gödel's encoding

Another popular way to encode $(x_1, \ldots, x_n)$ is the following:

$$p_0^{x_1+1} \times \ldots \times p_{n-1}^{x_n+1}$$

where $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, …, p_i is the i^{th} prime number.

A helpful notation to write the product above is $\prod_{i<n} p_i^{x_{i+1}+1}$, which is a notation similar to summation but for multiplication.

Because factorization into primes is unique, we guarantee that every string $(x_1, \ldots, x_n)$ has a unique code.

Remark 8 Either method of encoding allows us to recover the original string from its code. For example, if the code was defined by pairing, we can recover the original string using repeated use of the inverse of the pairing function, and if the code was done in what we call Gödel's style, then the original string can be recovered using prime factorization. There is one ambiguity here, but we ignored it above to not distract you from the core idea. This ambiguity pertains to the use of pairing in a context where strings have different lengths. Indeed, a given code can represent strings of different lengths. An easy resolution is to encode $(x_1, \ldots, x_n)$ as $\langle n, \langle\langle \ldots \langle\langle x_1, x_2\rangle, x_3\rangle, \ldots, x_{n-1}\rangle, x_n\rangle\rangle$.

Exercise 7 Recover the string s of natural numbers that has the code 315. Do it once assuming that the encoding was done using Cantor's pairing, and another time assuming the encoding was done in Gödel's style.

Generalization to any alphabet

Gödel's encoding was originally invented to encode logical formulas as natural numbers, which he used to prove his incompleteness theorems [3] The essence of the encoding is can be quickly explained by an example.

Example 2 **(Gödel's Encoding)**

Suppose we are encoding the strings made from the small English letters. In other words, our alphabet is

$$\{\text{a,b,c,}\ldots,\text{z}\}.$$

First we start with the shortest nonempty string, we encode each letter as a natural number, say:

$$gn(a) = 1,\ gn(b) = 2,\ gn(c) = 3,\ldots$$

where $gn(\ .\ )$ means the Gödel number of whatever is between the parenthesis.

Now consider the string "hello". We have

$$gn(\text{hello}) = 2^{gn(\text{h})}3^{gn(\text{e})}5^{gn(\text{l})}7^{gn(\text{l})}11^{gn(\text{o})} = 2^8 3^5 5^{12} 7^{12} 11^{15}.$$

This is obviously a massive number, which we do not care to compute here, but hopefully the idea has been demonstrated.

Example 3 **(Encoding Turing machines)**

Recall how we identified Turing machines with their programs (see Remark 7 and Definition 3). A Turing program such as

$$q_0\ 1\ q_0\ 1\ R$$
$$q_0\ b\ q_1\ 1\ R$$
$$q_1\ b\ q_2\ 1\ R$$

is a string of symbols. If we encode each symbol as a natural number, then we can encode the whole program as a natural number. Imagine this as giving the Turing machine a name, the name is a natural number. This becomes very useful when we work under the assumption that all our machines are built using the same Q and Γ.

[3] Soon, we will talk about Gödel's incompleteness theorems, simply because they are profoundly enchanting.

Exercise 8 Assume all our Turing machines use $\Gamma = \{b, 1\}$ and $Q \subset \{q_i : i \in \mathbb{N}\}$[4]. Suppose we agreed to the following encoding of symbols

$$gn(b) = 1, gn(1) = 2, gn(L) = 3, gn(R) = 4 \text{ and } gn(q_i) = 5 + i \text{ for all natural } i.$$

Compute the Gödel number of the program in the last example.

Notation

From now on, for any string s, we will write $\langle s \rangle$ to mean the encoding of s as a natural number. Of course this assumes we have chosen some encoding method to use throughout the book, it does not matter which one.

For example, whenever you see $\langle x_1, \ldots, x_n \rangle$, assume it is the natural number code obtained using Gödel's numbering. If you see $\langle M \rangle$ where M is some Turing machine, assume it is the Gödel code of the Turing program of M. If you see $\langle G \rangle$ where G is a graph, assume that $\langle G \rangle$ is the Gödel encoding of the adjacency matrix (e.g. dictionary in Python) that represents the graph.

Practice Problems

1. Suppose T is a machine which can use states from $q_i, i \in \mathbb{N}$, symbols from $b, 1$ with input/output convention as in Example **??**. Find a Turing machine T^* which computes exactly like T, except that, the output convention is exactly as the input convention. That is, on input n, the output must be $T(n)+1$ consecutive 1's with the head at the leftmost 1.

Chapter 3

Computability of Functions

3.1 A mechanical approach (Turing's Machine)

Definition 1 **(Partial Computable Function)**

A function $f\colon D \to \mathbb{N}$, with $D \subseteq \mathbb{N}^k$ (where $k \in \mathbb{N}$), is said to be *partial computable* (or *p.c.*) if there exists a Turing machine that computes f.

Since we believe the Church-Turing thesis, an easier definition to remember would be:

$$f \text{ is p.c.} \iff \text{There exists a computer program that computes } f.$$

The equivalence above means that the program halts (gives an output) exactly on the elements of the domain D. More clearly,

if $x \in D$, then, on input x, the program gives output $= f(x)$, and

if $x \notin D$, then, on input x, the program keeps running forever.

Remark 9 The word *partial* is used to emphasize that the domain D may not necessarily be everything (all of $\mathbb{N}$). That said, the definition does not prevent D from being equal to $\mathbb{N}$.

Definition 2 **(Computable Function)**

A function $f\colon \mathbb{N}^k \to \mathbb{N}$ (where $k \in \mathbb{N}$), is said to be *computable* if there exists a Turing machine that computes f.

Notice how similar the two definitions are. To make sure you understand the connections and differences between the two definitions, determine which of the following statements is **true** and which is **false**:

1. If f is a computable function, then f is a partial computable function.
2. Being computable is a special case of being partial computable.

3. If g is a partial computable function with domain $D \subsetneq \mathbb{N}$, then g is not computable.

Remark 10 The definitions above are about functions on natural numbers, but recall that every kind of data which a computing machine can handle is a natural number, whether it is a string of letters, a real number, or a structure such as a graph. This applies whether the machine uses binary, unary, or other kind of representation.

Remark 11 Similar to what we have in Remark 9, the word *total* can also be used to emphasize that the domain is all of $\mathbb{N}^k$ (or whatever domain of interest is being studied). *A computable function is a total partial computable function.*

Exercise 9 Consider the function $g = \{(2,3),(3,3),(5,4)\}$. Prove that g is partial computable.

Hint: a perfect solution must start by specifying the domain D. Then, write a program, or a Turing program, which computes g.

Exercise 10 Consider the function $h\colon \text{EVEN} \to \mathbb{N}$ given by $h(x) = 1$ for all $x \in \text{EVEN}$ where $\text{EVEN} = \{2x\colon x \in \mathbb{N}\}$. Prove that h is partial computable.

Exercise 11 Let EVEN be as in the previous exercise. Consider the function $f\colon \mathbb{N} \to \{0,1\}$ given by

$$f(x) = \begin{cases} 1 & \text{if } x \in \text{EVEN} \\ 0 & \text{otherwise} \end{cases}$$

1. Prove that f is computable.
2. Write f as a set of ordered pairs.

Exercise 12 Consider the function $f\colon \emptyset \to \mathbb{N}$.

1. Write f as a set.
2. Prove that f is p.c.

Notice how we said "**the**" function, not "**a**" function. Also notice that we did not describe any relation that defines how the function is being evaluated.

Turing's Thesis

Definition 1 (or Definition 2) is very clear, rigorous, and seems to capture how we humans understand computability. It is reasonable to believe that any function which we can compute using some process, can also be computed using some Turing machine (some Turing

program). That said, we have no actual proof that this is true. How could there even be a proof, while the word "process" is by nature vague? what do we even mean by proof? A mathematical one?

Accepting Definition 1 as a capture of the entire reality of computability is a philosophical matter - a state of faith. Perhaps it is safer to accept it as a mere part of reality which, as far as we can see, is enough to explore mathematics through a computational lens.

This state of faith is popularly known as *Turing's thesis*.

Exercise 13

We will say that a list (sequence) of natural numbers $A = \{a_i : i \in \mathbb{N}\}$ is computable if there exists a Turing machine which, when given the natural number i as an input, the machine gives a_i as an output.

Prove that there exists a Turing machine which, when given a natural number x as an input, the machine will output 1 if x is in the list, and the machine will not halt if x is not in the list.

In other words, prove that the following function is partial recursive:

$$f : A \to \mathbb{N},$$

such that, for all $x \in \mathbb{N}$,

$$f(x) = \left\{ \begin{array}{ll} 1 & \text{if } (\exists i \in \mathbb{N})[x = a_i] \\ \text{undefined} & \text{otherwise} \end{array} \right\}$$

Hint: Use Turing's thesis.

Request 1

Prove the statement in the previous exercise without appealing to Turing's thesis.

3.2 A mathematical approach (Gödel's Recursive Functions)

While the definitions in the previous section are fairly intuitive, there is another approach to define computability of functions without resorting to machines (Turing machines). Such an approach, even if less intuitive, it is still valuable to discuss.

Let us reflect again on the most basic abilities of mind. First, think of the following:

Nothing vs. **Something**

and if there is something(s), we can

make a choice.

These concepts are captured through the following *initial* functions:

the **Zero function**: $Z : \mathbb{N} \to \mathbb{N}$, given by $Z(x) = 0$ for all $x \in \mathbb{N}$;

the **Successor function**: $S : \mathbb{N} \to \mathbb{N}$, given by $S(x) = x + 1$ for all $x \in \mathbb{N}$;

the **Projection functions**: This is a family of functions each of which describes a choice from multiple.

Given k-many natural numbers, choose the i^{th} $(i \leq k)$. So, for natural numbers k and i, with $i \leq k$, there is a function

$$P\colon \mathbb{N}^k \to \mathbb{N}, P(n_1, \ldots, n_k) = n_i.$$

To be proper, we label P with some indices and write it as P_i^k. The superscript is the size (dimension) of the allowable input, whereas the subscript is the index of the component to be chosen (projected).

The family of projections is

$$Proj = \{P_i^k\colon k \text{ is a positive natural number, and } 1 \leq i \leq k\}.$$

All the initial functions together is the following collection:

$$Init = \{Z, S\} \cup Proj.$$

Secondly (back to reflecting on the basic abilities of mind), **think of how to use the initial functions to build more complex functions**.

Probably, the first and most natural thing one could think of doing is to use **composition**. For example, one can compose the successor function S with the zero function Z to obtain the new function $f = S \circ Z$. More clearly, $f\colon \mathbb{N} \to \mathbb{N}$, given by $f(x) = S(Z(x))$; which is clearly the constant function $f(x) = 1$.

It is easy to see that, in a similar way, one can build any constant function $g\colon \mathbb{N} \to \mathbb{N}$, $g(x) = c$, where $c \in \mathbb{N}$. All that is needed is to apply composition c-many times: $g = \underbrace{S \circ \ldots \circ S}_{c\text{-many}} \circ Z$.

It is useful to think of composition as an operator $\circ$ which, from two given functions f, g it builds the function $f \circ g$. More clearly, if $f, g\colon \mathbb{N} \to \mathbb{N}$, then

$$\circ(f, g)\colon \mathbb{N} \to \mathbb{N}$$

such that

$$\circ(f, g)(x) = f(g(x)).$$

Or, more generally, from given functions $f, g_1, \ldots, g_m$ where

$$f\colon \mathbb{N}^m \to \mathbb{N}, \text{ and } \forall i \in \{1, \ldots, m\}, g_i\colon \mathbb{N}^k \to \mathbb{N},$$

we have

$$\circ(f, g_1, \ldots, g_m)\colon \mathbb{N}^k \to \mathbb{N}$$

such that

$$\circ(f, g_1, \ldots, g_m)(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k)).$$

Exercise 14 Use initial functions to define the identity function $I\colon \mathbb{N} \to \mathbb{N}$, $I(x) = x$.

Exercise 15 Use the initial functions to define (or build) the function Plus4$\colon \mathbb{N} \to \mathbb{N}$, Plus4$(x) = x + 4$.

Exercise 16 Try to use initial functions to define addition. That is, to define the function

$$\text{Plus}\colon \mathbb{N}^2 \to \mathbb{N}, \ \text{Plus}(x_1, x_2) = x_1 + x_2.$$

In the last exercise, the word "Try" is used for a reason. It is very possible that you have thought of

$$S^{x_1}(x_2) = \underbrace{S \circ \ldots \circ S}_{x_1\text{-many}}(x_2).$$

You know how to use composition repeatedly as many times as needed when x_1 changes, but the function itself has no rule to figure that out when x_1 is not constant. Defining $+$ this way hides more than just composition. A new building rule is needed.

The new rule is called **Primitive Recursion**. As the name suggests, it allows for recursion, which is a primitive mind ability if you think about it. The ability to define something using previous (possibly simpler) versions. In what follows, we describe primitive recursion as a building rule (or procedure). The description is very general, and may be difficult to understand at first.

In a way similar to how we defined the operator $\circ$ above, we now define a new operator ρ that denotes the primitive recursion building rule. Given functions $g\colon \mathbb{N}^k \to \mathbb{N}$ and $h\colon \mathbb{N}^{k+2} \to \mathbb{N}$, we have

$$\rho(g,h)\colon \mathbb{N}^{k+1} \to \mathbb{N}$$

such that, for all $x_1, \ldots, x_k \in \mathbb{N}$,

$$\rho(g,h)(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k),$$

and

$$\rho(g,h)(S(n), x_1, \ldots, x_k) = h(n, \rho(g,h)(n, x_1, \ldots, x_k), x_1, \ldots, x_k).$$

One may understand this definition by imagining the functions g, h collaborating to describe behavior of the function $\rho(g,h)$ recursively. As if we are recursively defining a sequence $a_0, a_1, \ldots$ where $a_i = \rho(g,h)(i, x_1, \ldots, x_k)$.

The function g describes the "base level" of $\rho(g,h)$, level 0, this is why $\rho(g,h)(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$. The function h describes how level $n+1$ (which is $\rho(g,h)(S(n), x_1, \ldots, x_k)$) is defined by making use of the previous level, level n (which is $\rho(g,h)(n, x_1, \ldots, x_k)$).

Notice that level $n+1$, $\rho(g,h)(S(n), x_1, \ldots, x_k)$, is defined using n, $x_1, \ldots, x_k$, and the previous level $\rho(g,h)(n, x_1, \ldots, x_k)$. This allows more freedom using recursion, and naturally makes more sense than using $\rho(g,h)(n, x_1, \ldots, x_k)$ alone (e.g. $h(\rho(g,h)(n, x_1, \ldots, x_k))$). As you can see, $n, x_1, \ldots, x_k$ are already explicit variables in the definition of $\rho(g,h)$, so why not allow them explicitly in h.

To better understand primitive recursion, consider the following example. The function Plus of Exercise 16 can be defined using primitive recursion as follows:

$$\text{Plus} = \rho(g,h)$$

where

$$g = P^1_1,$$

and $h\colon \mathbb{N}^3 \to \mathbb{N}$ such that

$$h(n, x, x_1) = S(x).$$

These choices of g, h work because:

At level 0, for any $x_1 \in \mathbb{N}$, $\rho(g,h)(0, x_1) = g(x_1) = P^1_1(x_1) = x_1$, which is exactly how we want it to be because $0 + x_1 = x_1$.

At level 1, for any $x_1 \in \mathbb{N}$, $\rho(g,h)(1,x_1) = h(1,\rho(g,h)(0,x_1),x_1) = h(1,x_1,x_1) = S(x_1) = x_1 + 1$, which is, again, exactly how we want it to be.

At level 2, for any $x_1 \in \mathbb{N}$, $\rho(g,h)(2,x_1) = h(2,\rho(g,h)(1,x_1),x_1) = h(2,x_1+1,x_1) = S(x_1+1) = x_1 + 2$, which is, again, exactly how we want it to be.

.

.

.

and so on. Of course this is not formal, but can easily be formalized using a proof by induction.

For a fixed natural x_1, one can show by induction that for all natural n, $\text{Plus}(n,x_1) = \rho(g,h)(n,x_1)$. This is equivalent to saying that, for all $x_1, x_2 \in \mathbb{N}$, $\text{Plus}(x_1,x_2) = \rho(g,h)(x_1,x_2)$.

For the sake of reviewing Mathematical Induction, we include a detailed proof here.

The statement we want to prove is the following:

$$(\forall n \in \mathbb{N})[\text{Plus}(n,x_1) = \rho(g,h)(n,x_1)],$$

where x_1 is an arbitrary natural number.

Proof:

First let us use $P(n)$ to denote the open sentence $\text{Plus}(n,x_1) = \rho(g,h)(n,x_1)$.

Base Case: We want to prove the statement $P(0)$, which is the following equation:

$$\text{Plus}(0,x_1) = \rho(g,h)(0,x_1).$$

To prove that an equation is true, we need to show that the LHS and the RHS have the same value.

The LHS, $\text{Plus}(0,x_1)$, is equal to $0 + x_1 = x_1$.

The RHS, $\rho(g,h)(0,x_1)$, is equal to $g(x_1) = P_1^1(x_1) = x_1$.

The two sides are equal, and therefore the statement $P(0)$ is true.

The Induction Hypothesis: Assume that $P(n)$ is true for some arbitrary $n \in \mathbb{N}$ [1].

The Induction Step: Now we use the induction hypothesis, which is the assumption that $P(n)$ is true, to prove that $P(n+1)$ is true. In other words, we want to show that

$$P(n) \Longrightarrow P(n+1).$$

The assumption that $P(n)$ is true (the induction hypothesis) means that $\text{Plus}(n, x_1) = \rho(g,h)(n, x_1)$.

We want to show that:

$$\text{Plus}(n+1, x_1) = \rho(g,h)(n+1, x_1).$$

The LHS $= \text{Plus}(n+1, x_1) = (n+1) + x_1$, which is equal to $(n + x_1) + 1$ [2].

The RHS $= \rho(g,h)(n+1, x_1) = \rho(g,h)(S(n), x_1) = h(n, \rho(g,h)(n, x_1), x_1) = S(\rho(g,h)(n, x_1))$.

From the induction hypothesis, $\rho(g,h)(n, x_1) = \text{Plus}(n, x_1)$. As we know, $\text{Plus}(n, x_1) = n + x_1$, and so we have $\rho(g,h)(n, x_1) = n + x_1$.

Therefore, The RHS $= S(\rho(g,h)(n, x_1)) = S(n + x_1) = (n + x_1) + 1$, which is exactly the same as the LHS.

From the principle of mathematical induction, the proof is now complete!

Primitive Recursive Functions

So far, we have the initial functions Z, S, and the P^i_j's, and we have the building rules $\circ, \rho$. Using these, we can build so many interesting functions. The collection of all functions one can build this way is called the class of Primitive Recursive Functions (abbreviated as: PRIM).

It is not hard to believe[3] that every function in PRIM can be defined using a Turing machine. This makes

$$\text{PRIM} \subseteq \text{Partial Computable Functions}.$$

Moreover, it is also easy to believe that all functions in PRIM are total, which makes

$$\text{PRIM} \subseteq \text{Computable Functions}.$$

[1] After making such an assumption, you should think of n as a fixed value, and no longer as a variable. This is why some people prefer to use a new symbol, say k or n_0.

[2] This equality follows from the commutativity and associativity of $+$ on $\mathbb{N}$

[3] One can prove this formally using a generalized version of induction proofs. See the appendix.

Since all functions in PRIM are total, it is then obvious that

$$\text{PRIM} \subsetneq \text{Partial Computable Functions}.$$

It is also true (but not obvious) that PRIM $\subsetneq$ Computable Functions. In other words, not every total function that can be defined using a Turing machine is a primitive recursive function.

The fact that PRIM $\subsetneq$ Partial Computable Functions tells us that some basic ability of mind is not captured using Z, S, the P_j^i's, $\circ$, and ρ. What could be missing here? Hint: Think of an operation which, if defined using an algorithm, the algorithm may not halt. The answer is in the next section.

Exercise 17 Write a Turing Machine for each one of the functions below:

1. The function $Z : \mathbb{N} \to \mathbb{N}$ given by $Z(n) = 0$ for every $n \in \mathbb{N}$.
2. The function $S : \mathbb{N} \to \mathbb{N}$ given by $S(n) = n + 1$ for every $n \in \mathbb{N}$.
3. The function $P_2^5 : \mathbb{N}^5 \to \mathbb{N}$ given by $P_2^5(n_1, \ldots, n_5) = n_2$ for every $n_1, \ldots, n_5 \in \mathbb{N}$.

Exercise 18 Prove that the function which gives the minimum of two natural numbers is in PRIM.

(General) Recursive Functions

Have you found out which ability is missing in PRIM? The answer is **performing a search**. None of the functions or operations in PRIM allows for searching an element in a list (possibly infinite), which is a very basic mental ability.

Wait a second, is it a basic ability? Can we use a Turing machine to search an element in a given list?

The answer is **yes**, see Exercise 13. Of course a search makes sense only under the assumption that the list is computable as explained in Exercise 13.

Performing a search can be captured via the following new building rule, which we denote by μ, and call it *minimization*. Given a function $f \colon \mathbb{N}^{k+1} \to \mathbb{N}$, we have

$$\mu(f) \colon \mathbb{N}^k \to \mathbb{N}$$

such that, for all $x_1, \ldots, x_k \in \mathbb{N}$,

$$\mu(f)(x_1, \ldots, x_k) = z$$

where

$$f(z, x_1, \ldots, x_k) = 0 \text{ and } (\forall z' < z)[f(z', x_1, \ldots, x_k) \neq 0].$$

In other words, $\mu(f)(x_1, \ldots, x_k)$ is the smallest natural value z that makes the following equation true:

$$f(z, x_1, \ldots, x_k) = 0.$$

To clarify things more, imagine that you had some search problem where you wanted to find a value that satisfies a particular mathematical statement. Then, you managed to formulate this search problem as solving some equation that has one unknown z and some parameters $x_1, \ldots, x_k$: $f(z, x_1, \ldots, x_k) = 0$.

The collection of all functions one can build using Z, S, P^i_j's and $\circ, \rho$, and μ is called the class of General Recursive Functions. The word "General" is sometimes omitted.

One can prove that

$$\text{General Recursive Functions} = \text{Partial Computable Functions}.$$

The inclusion $\subseteq$ is easy to believe. The other direction $\supseteq$ is not too obvious, but can be proved using induction.

Remark 12 General recursive functions are basically the ones for which you can write a recursive program, where are Turing's partial computable functions are the ones for which you can write an iterative program.

Church's Thesis

The class of (general) recursive functions is defined in a very clear and rigorous manner. Similar to Turing's thesis, there is a state of faith that (general) recursive functions are all the functions we can compute by some process. This state of faith is known as *Church's thesis*.

Recall that the following is a provable fact: General Recursive Functions = Partial Computable Fun
This is one piece of evidence to support the faith.

It is common to see either thesis referred to as the *Church-Turing thesis*, with the two names together.

Practice Problems

1. Let $f\colon \mathbb{N} \to \mathbb{N}$ be a computable function. Suppose that f is bijective, and so an inverse function exists. Prove that the inverse function of f must be computable.
2. Let p_n denote the n^{th} prime number with $p_0 = 2$. Prove that the function $f\colon \mathbb{N} \to \mathbb{N}$ such that $f(n) = p_n$ is a (general) recursive function.

3. Let $\mathbb{N}^*$ be the set of all nonempty finite sequences of natural numbers. That is, $\mathbb{N}^* = \cup_{k=1}^{\infty} \mathbb{N}^k$. Consider the following function

$$p\colon \mathbb{N}^* \to \mathbb{N}$$

such that

$$p(a) = \prod_{i<|a|} p_i^{a_i}$$

where p_i are as in the previous exercise. For example, $p((3,2,5)) = 2^3 3^2 5^5$.

Prove that p is computable.

4. Let p_n be as in the previous problem. For a natural number m, let $(m)_i$ denote the exponent of p_i in the prime factorization of m. For example, $(72)_1 = 2$ because $72 = 2^3 3^2$. Prove that the function $g\colon \mathbb{N}^2 \to \mathbb{N}$ given by $g(m,i) = (m)_i$ is a (general) recursive function.

5. Prove that multiplication is primitive recursive.

6. Prove that the Cantor pairing function is primitive recursive.

7. Recall the functions π_1, π_2 which give the components of the inverse of Cantor's pairing. Prove that π_1 and π_2 are primitive recursive.

8. (**Course-of-values Recursion**) First, let us introduce a new helpful definition. Let f be a function defined using primitive recursion, say $f = \rho(g,h)\colon \mathbb{N}^k \to \mathbb{N}$. Consider the following function, which we call the *history of* f:

$$\widehat{f}\colon \mathbb{N}^k \to \mathbb{N},$$

$$\widehat{f}(n,\vec{x}) = \langle f(0,\vec{x}), \ldots, f(n,\vec{x})\rangle,$$

where the notation [4] $\vec{x}$ is just an easy way to write $x_1, \ldots, x_k$.

Now let us define a new operator, which we denote $\tilde{\rho}$. Given $h\colon \mathbb{N}^{k+2} \to \mathbb{N}$ and $g\colon \mathbb{N}^k \to \mathbb{N}$. Let

$$\tilde{\rho}(g,h)\colon \mathbb{N}^{k+1} \to \mathbb{N},$$

such that

$$\tilde{\rho}(g,h)(0,\vec{x}) = g(\vec{x}),$$

$$\tilde{\rho}(g,h)(S(n),\vec{x}) = h(n, \widehat{\rho(g,h)}(n,\vec{x}),\vec{x}).$$

The last expression can also be written as

$$\tilde{\rho}(g,h)(S(n),\vec{x}) = h(n, \langle \rho(g,h)(0,\vec{x}), \ldots, \rho(g,h)(n,\vec{x})\rangle, \vec{x}).$$

In simple words, the operator $\tilde{\rho}$ does a very similar job to ρ, but by using all the values $\rho(g,h)(0,\vec{x}), \ldots, \rho(g,h)(n,\vec{x})$ instead of using only $\rho(g,h)(n,\vec{x})$. In other words, it is a recursion using (possibly) all the previous values, not just the last one.

[4] We will use this kind of notation from now on throughout the book.

Now, suppose that g, h are from PRIM, and that the encoding $\langle \ldots \rangle$ is also in PRIM. Prove that $\tilde{\rho}(g, h)$ is also in PRIM.

This tells us that, even though $\tilde{\rho}$ seems on surface to be a more powerful rule than ρ, in reality they are equivalent as $\tilde{\rho}$ does not allow for building more functions than ρ does.

Chapter 4

Decision Problems

4.1 Computability of Sets (Decidability)

In this chapter we are concerned with Yes/No questions. Computationally, such questions can be studied as functions from $\mathbb{N}$ to $\{0, 1\}$. Another equivalent way to computationally formalize a Yes/No question is to make it a membership question about a subset of $\mathbb{N}$.

To clarify, given $A \subseteq \mathbb{N}$ and any element $n \in \mathbb{N}$, we have the question:

$$\text{Does } n \text{ belong to } A \text{ ?}$$

Answering this Yes/No question is equivalent to evaluating the *indicator* function [1] of A at n, that is, evaluate $I_A(n)$ where

$$I_A \colon \mathbb{N} \to \mathbb{N},$$

$$I_A(x) = \left\{ \begin{array}{ll} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{array} \right\}$$

Definition 1 **(Decidable Set)**

A set A of natural numbers ($A \subseteq \mathbb{N}$) is said to be *Decidable* (or *Computable*) if I_A is computable (recall Definition 2).

Informally, the definition is saying that A is decidable if there exists a computer or a process (a Turing program) that can answer membership questions about A. In other words, that computer can decide, for any given natural number n, whether n belongs to A or not.

It is natural to see that the concept of decidability can be generalized to include sets A that are not necessarily made from natural numbers, but from a different universe $\mathcal{U}$. Just keep in mind that:

[1] It is also common to call it the *characteristic* function of the set A.

1. the elements of $\mathcal{U}$ must be encodable as natural numbers (or as strings of some finite alphabet), and
2. the codes of the elements of $\mathcal{U}$ form a decidable subset of $\mathbb{N}$.

Without the first requirement, it would be impossible to computationally tackle decidability questions in $\mathcal{U}$. **What about the second requirement?**

To see why the second requirement is there, first try to create a version of Definition 1 for subsets of $\mathcal{U}$. You want your definition to formalize the following idea:

A set A is decidable if there exists a process (Turing machine) M_A which can answer membership questions about A.

The machine M_A should be able to give an answer for any input (any natural number or string from its tape alphabet). Otherwise, it will not be considered capable of answering membership questions about A. In other words, M_A has to act like a total computable function.

Now imagine M_A is given some input n. That n may or may not be the code of some element in $\mathcal{U}$. The machine M_A will have to, first, decide whether n is in $\mathcal{U}$ or not. If n is not in $\mathcal{U}$, then M_A will immediately decide that n is not in A (because $A \subseteq \mathcal{U}$)[2]. If n is in $\mathcal{U}$, then M_A will decide whether n is in A or not.

The following exercise should enhance your understanding of what we have just discussed:

Exercise 19

Let $f\colon D \to \mathbb{N}$ be a partial computable function such that D is a decidable subset of $\mathbb{N}$. Prove that $f \circ I_D$ is computable.

Let us call universes that fulfill requirements 1 and 2 *acceptable*. Here is a generalization we suggest:

Definition 2 **(Decidability Generalized)**

In an acceptable universe $\mathcal{U}$, a set $A \subseteq \mathcal{U}$ is decidable if the following function is partial computable:

$$I_A\colon D \to \{0,1\},$$

$$I_A(x) = \left\{ \begin{array}{ll} 1 & \text{if } x \in A \\ 0 & \text{if } x \in D \setminus A \end{array} \right\}$$

where D is the set of codes of the elements in $\mathcal{U}$.

To make things easier, most of the time we will talk about I_A as a function with domain $\mathcal{U}$ instead of D because transitioning between $\mathcal{U}$ and D (the encoding/decoding process) is assumed to be known and computationally easy.

[2]The machine can be designed to give some kind of error message to indicate that n is not from the universe where A lives.

Examples

Here are some Yes/No questions which a computer can handle:

1. Is 344 an even number?
2. Does the equation $x^2 + y^3 = 12$ have a solution in $\mathbb{Z}$?
3. Given some (directed) graph G, with two of its vertices specified, say s and t, is there a path from vertex s to vertex t ?

Each of these questions is asked within the context of some universe, and it is about a subset of that universe. For example, the universe of the first question can be $\mathbb{Z}$, and the subset is the $E = \{x \in \mathbb{Z} \colon x \text{ is even }\}$.

For the second question, the universe could be

$$\{p(x, y) = 0 \colon p(x, y) \text{ is a polynomial with integer coefficients in the two variables } x, y\},$$

and then the subset is

$$\{p(x, y) = 0 \colon$$
$$p(x, y) \text{ is a polynomial with integer coefficients in the two variables } x, y \text{ and}$$
$$\exists a, b \in \mathbb{Z}, p(a, b) = 0\}.$$

Notice how for each of the questions above, a universe gives a context for the question, and there could be more than one "sensible" universe.

Question 3 What is a sensible universe for the fourth question?

Definition 3 **(Decision Problems)**

A decision problem on an acceptable universe $\mathcal{U}$ is just a subset of $\mathcal{U}$. Without loss of generality, we can define a decision problem as a subset of $\mathbb{N}$.

Formulating decision problems as subsets makes it easier to rigorously study them computationally. For example, suppose we want to define the class of decision problems which a computer can solve. Then, we have Definition 1.

Question 4 Is there a set $A \subseteq \mathbb{N}$ which is not decidable? What does it look like?

The **existence** of such a set can be easily proven using a cardinality argument. Hint: there is only countably many decidable sets, while there is uncountably many subsets of $\mathbb{N}$.

But, we can also prove existence by finding an example of such a set, which you will do in the next exercise.

Exercise 20 Recall how every Turing machine M has a Gödel number, denote it $gn(M)$. Assume without loss of generality that all our machines correspond to functions on $\mathbb{N}$. Consider now the set $K = \{gn(M) : M \text{ halts when given } gn(M) \text{ as an input}\}$. Prove that K is not decidable.

Hint: Use contradiction. Assume that K is decidable, which means that there exists a Turing machine M_K which computes the indicator function I_K. Then, the natural number $gn(M_K)$ belongs to K and does not belong to K at the same time.

Remark 13 The set K in the last exercise is known as *Kleene's halting set*. If we think of the set K as a decision problem, it is unsolvable (undecidable).

Exercise 21 Let $H = \{(e, x) :$ the Turing machine with Gödel number e halts on input $x\}$. The set H is known as the *halting set*. Prove that H is not decidable.

We finish this chapter by pointing out that, with the concept of acceptable universe in mind, we can talk about computability and partial computability for functions from some acceptable universe to another acceptable universe. In what follows we use en to denote an example of a computable encoding (e.g. Gödel's).

Definition 4 **(Computability of Functions - Generalized)**

Let $\mathcal{U}_1$ and $\mathcal{U}_2$ be two acceptable universes. Let $D \subseteq \mathcal{U}_1$. A function $f : D \to \mathcal{U}_2$ is a partial computable if $en \circ f \circ en^{-1}$ is partial computable function from the set of codes of D to the set of codes of $\mathcal{U}_2$.

If $D = \mathcal{U}_1$, we say f is computable from $\mathcal{U}_1$ to $\mathcal{U}_2$.

Request 2 Reflect on the possibility of having a finite (possibly empty) acceptable universe $\mathcal{U}_1$ in the previous definition.

Practice Problems

1. Let $f : \mathbb{N} \to \mathbb{N}$ be a strictly increasing computable function. Show that $range(f)$ is a decidable set.
2. Prove that a function f is computable according to Definition 2 if and only if it is decidable as a set according to Definition 1.
3. Prove that any finite set is decidable.
4. Prove that the union of any finite number of decidable sets is also decidable.
5. Prove that the intersection of any finite number of decidable sets is also decidable.
6. Prove that the countable union of decidable sets is not necessarily decidable.
7. Prove that the Cartesian product of two decidable sets is also decidable.

4.2 Enumerability of Sets (Recognizability)

We will say a Turing machine *recognizes* an input if it halts on that input. In other words, if the input is valid and makes sense to the machine.

Definition 5 **(Recognizable Set)**

A set A of natural numbers is *recognizable* if there exists a Turing machine M_A which halts exactly on the elements of A.

Equivalently, A is recognizable if $A = dom(\varphi)$ for some partial computable function φ.

In other words, A is the set of values recognizable by some Turing machine.

Similar to decidability, recognizability can be defined for sets that are not from natural numbers. A recognizable set may be a subset of some acceptable universe, using Definition 4.

Exercise 22 Prove that every decidable set is recognizable.

Exercise 23 Let f be a computable function. Prove that $range(f)$ is recognizable.

Exercise 24 Prove that any nonempty[3] recognizable set is the range of some computable function.

The statements in the last two exercises are the reason why recognizable sets are also known as *computably enumerable* sets or *listable* sets. To clarify, if f is computable, and $A = range(f)$, then $A = \{f(0), f(1), , \ldots\}$, as if the elements of A have been computably enumerated: the computable procedure f labeled an element in A by 0, then labeled another element by 1, and so on. Another way to see enumerability/listability, consider the following program which prints the elements of A (possibly with repetition):

$$\text{For } i \in \mathbb{N}: \text{ Print } f(i).$$

Another popular name for recognizable sets is *semi-decidable* sets. The reason for calling them so becomes obvious when you look at a recognizable set as a decision problem. If $A \subseteq \mathbb{N}$ is recognizable, and $x \in \mathbb{N}$ is arbitrary, then the question "Does x belong to A?" is certainly answerable by a computer if x is in A (just run M_A from Definition 5 on input x). However, if x is not from A, then certainly M_A will not tell us anything because it will not halt on x, and there is no guarantee that any other Turing machine can confirm that $x \notin A$. The following exercise will help you understand semi-decidability more:

Exercise 25 Let A be a set of natural numbers. Prove that A is recognizable if and only if there exists a partial recursive function J_A such that:

$$J_A(x) = \left\{ \begin{array}{ll} 1 & \text{if } x \in A \\ \text{undefined or halts with a value different from 1} & \text{if } x \in \mathbb{N} \setminus A \end{array} \right\}$$

You learnt from Exercise 22 that every decidable set is recognizable. However, the converse is not true:

Exercise 26 Show that the sets K and H from Exercises 20 and 21 are recognizable.

We finish this chapter by reminding you that recognizable, semi-decidable, computably enumerable (abbreviated as c.e.), and listable; all these words mean the exact same thing.

Practice Problems

1. Let R be a binary relation on $\mathbb{N}$, and let A be the following set

$$A = \{x \in \mathbb{N} \colon \exists y, R(x, y)\}.$$

   Recall that $R(x, y)$ is just a short way to write $(x, y) \in R$. Assume that R is computable. Prove that A is c.e.

2. Prove that the union of any finite number of recognizable sets is also recognizable.

3. Prove that the intersection of any finite number of recognizable sets is also recognizable.

4. Prove that the countable union of recognizable sets is not necessarily recognizable.

5. Let A be a recognizable set of natural numbers. Suppose that $\mathbb{N} \backslash A$ is also recognizable. Prove that A is decidable.

Chapter 5

Comparing Decision Problems

Some decision problems are more unsolvable than others. In this chapter, we define some binary relations on the collection of all decision problems. These binary relations will act in a way similar to $\leq$ on numbers (order relations).

5.1 Many-one reducibility

Definition 1 **(m-reducibility)**

Let A and B be sets of natural numbers. We say that A is m-reducible (or many-one reducible) to B if there exists a computable function f such that:

$$(\forall x \in \mathbb{N})[x \in A \iff f(x) \in B].$$

Or more generally, if A is a subset of an acceptable universe $\mathcal{U}_1$, and B is a subset of an acceptable universe $\mathcal{U}_2$. We say that A is m-reducible (or many-one reducible) to B if there exists a computable function $f : \mathcal{U}_1 \to \mathcal{U}_2$ such that:

$$(\forall x \in \mathcal{U}_1)[x \in A \iff f(x) \in B].$$

We write $A \leq_m B$, and we call f an m-reduction of A to B.

The wording *reducible to* in all reducibility definitions (we will see more reducibilities soon) can be misleading in the sense that it can make one think that the decision problem A was simplified (reduced) to problem B. On the contrary, problem B is in some sense more difficult than problem A as the following exercise shows:

Exercise 27 Suppose that $A \leq_m B$. Prove that if B is decidable, then A is decidable. However, the converse is not necessarily true (find a counter-example).

The exercise says that if B is a computably solvable decision problem, then A must also be a computably solvable decision problem. As if the ability to solve B implies the ability to solve A, but not the other way around.

Exercise 28 Prove that: If B is recognizable, and $A \leq_m B$, then A is also recognizable.

Exercise 29 Prove that every recognizable set is m-reducible to the halting set.

Remark 14 Notice that the definition of m-reducibility (and any concept of reducibility we will see) does not require that A and B are decidable. They can be sets of any level of unsolvability there is.

Remark 15 When an m-reduction from A to B is injective, we call it a 1-reduction, and write $A \leq_1 B$.

5.2 Turing's reducibility

The next two definitions pave the road to another concept of reducibility. It is more natural, but a bit more challenging to formalize than m-reducibility.

Definition 2 **(Relative Computability - Kleene)**

Recall how f is partial computable if it can be defined from the initial functions, $Init$, using composition, primitive recursion, and minimization. We will say that f is *partial computable relative to* A (some set of natural numbers) if f can be defined from $Init \cup \{I_A\}$ using composition, primitive recursion, and minimization.

Intuitively, f is partial computable relative to A if there exists a process (computer program) to compute f using I_A. More clearly, to compute $f(n)$ for some $n \in \mathbb{N}$, the process is allowed to use $I_A(m)$ for any $m \in \mathbb{N}$. As if the process has access to a black box that can decide A.

One way to formalize this intuition mechanically is by using what is called *oracle Turing machines*. These machines differ from the regular Turing machines, mainly in how the transition function works. There is not a single way to define such machines, but here is one:

Definition 3 **(Oracle Turing machine)**

An oracle Turing machine with oracle A (some set of natural numbers) differs from a regular Turing machine by allowing a new kind of instructions. The new kind takes the form "$q_i\ s\ q_j\ q_l$", which means that: if the head is at state q_i reading s, then it will transition to state q_j or to state q_l depending on what the oracle decides about the number reached on the tape. To explain this part, suppose the tape has n-many 1's when the head is at state q_i reading s. Then, if $n \in A$, the state of the head will change to q_j, and if $n \notin A$, the state of the head will change to q_l.

Example 1

Consider the following oracle machine

$$q_0\ 1\ q_0\ 1\ R$$

$$q_0\ b\ q_1\ 1\ R$$

$$q_1\ b\ q_2\ q_3$$

$$q_2\ b\ q_2\ 1\ R$$

$$q_3\ b\ q_3\ b\ R$$

Assume that the input/output convention is as in Example **??**, and assume that the oracle decision is based on the number of 1's on the tape when the state is q_1. It is not difficult to see that, if this machine has oracle A, then the function being computed is the following:

$$f(x) = \left\{ \begin{array}{ll} x+3 & \text{if } x+2 \in A \\ x+2 & \text{if } x+2 \in \mathbb{N} \setminus A \end{array} \right\}$$

At this point you can think of oracles as a source of external knowledge, a superpower. Now we have the following definition:

Definition 4 **(Relative Computability - Turing)**

A function f is partial computable relative to A if there exists an oracle Turing machine with oracle A which computes f.

One can prove the equivalence of definitions 2 and 4. We also have a relativized version of the Church-Turing thesis:

Definition 5 **(The Relativized Church-Turing thesis)**

Any formalization equivalent to Definition 2 or Definition 4 is equivalent to the intuition that "f is computable from A".

Definition 6 **(Turing's reducibility)**

Let A and B be sets of natural numbers (which can be generalized to acceptable universes). We say that A is *Turing-reducible to* B (in symbols, $A \leq_T B$) if I_A is computable relative to B.

The definition can also be rephrased to be about functions. That is, for functions f, g on the naturals (or any acceptable universes), we have $f \leq_T g$ if f (as a set) is computable relative to g (as a set).

It is not hard to see that m-reducibility is stronger than Turing's reducibility:

Exercise 30 Prove that, for any $A, B \subseteq \mathbb{N}$: if $A \leq_m B$, then $A \leq_T B$.

It is natural at this point to have: "decidable relative to A" or "recognizable relative to A". The meaning of each is obvious, where we just use computability relative to A instead of regular computability. For example, X is recognizable relative to A if X is the domain of some function φ which is partial computable relative to A (recall Definition 5).

Remark 16 It is common to use A-computable, A-decidable, A-recognizable, A-c.e. etc. to mean the adjective relativized to A.

Practice Problems

1. Unlike m-reducibility, if $A \leq_T B$ and B is recognizable, then A is not necessarily recognizable. Give an example of sets A, B to prove the previous statement.
2. Prove that the relations $\leq_m$ and $\leq_T$ are transitive.
3. Let $A \subseteq \mathbb{N}$ be a decidable set. Prove that, for any set $X \subseteq \mathbb{N}$, $A \leq_T X$. Is the same true for $\leq_m$?
4. Let A, B, and X be sets of natural numbers. Suppose that $B \leq_T A$ and that X is recognizable relative to B. Prove that A is recognizable relative to A.
5. Construct a set of natural numbers which is A-recognizable but not A-decidable.
6. Let K be Kleene's halting set, and let H be the halting set. Prove that K is computable relative to H.
7. Consider the binary relation $\equiv_T$ on $\mathcal{P}(\mathbb{N})$ such that:

   $$A \equiv_T B \iff A \leq_T B \wedge B \leq_T A.$$

   In other words,

   $$\equiv_T = \{(A, B) \in \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \colon A \leq_T B \wedge B \leq_T A\}.$$

   Prove that $\equiv_T$ is an equivalence relation on $\mathcal{P}(\mathbb{N})$. This is why $\equiv_T$ is known as Turing's equivalence.
8. Let $\equiv_m$ be defined similar to $\equiv_T$ but with m-reducibility instead of Turing's reducibility. Prove that $\equiv_m$ is an equivalence relation on $\mathcal{P}(\mathbb{N})$.
9. Let A be a set of natural numbers. Define the following, which is known as the *Turing degree of* A:

   $$deg(A) = \{X \in \mathcal{P}(\mathbb{N}) \colon X \equiv_T A\}.$$

   Notice that $deg(A)$ is just the equivalence class (of the relation $\equiv_T$) that contains A. Let $\mathcal{D}_T$ be the set of all Turing degrees. What is the cardinality of $\mathcal{D}_T$?
10. Consider the following binary relation on $\mathcal{D}_T$: For $\mathbf{a}, \mathbf{b} \in \mathcal{D}_T$, we have $\mathbf{a} \leq \mathbf{b}$ if there exists some $A \in \mathbf{a}$ and some $B \in \mathbf{b}$ such that $A \leq_T B$. Prove that this binary relation is well-defined on $\mathcal{D}_T$.
11. Prove that $\leq$ from the previous question is a partial order on $\mathcal{D}_T$.
12. Show that all decidable sets are Turing equivalent. This means they all fall in the same class which is $deg(\emptyset)$.
13. Given two sets of natural numbers A, B, one can define $A \oplus B$, which is known as the join of A, B as follows:

    $$A \oplus B = \{2x \colon x \in A\} \cup \{2x + 1 \colon x \in B\}.$$

    Prove that:

    (a) $A \leq_m A \oplus B$
    (b) $B \leq_m A \oplus B$
    (c) for all C such that $A \leq_m C$ and $B \leq_m C$, we have that $A \oplus B \leq_m C$

Chapter 6

Index Sets and Some Important Theorems

Notation

Before we proceed to more advanced topics, we will learn some helpful notation[1].

Partial Recursive Function number e: We learnt that every Turing machine has a Gödel number. We also learnt that every Turing machine corresponds to a partial recursive function. We will use the notation φ_e to denote the partial recursive function associated with the Turing machine that has Gödel number e, and we will refer to φ_e as the e^{th} partial computable function (or the partial computable function with index e).

Halts (or converges) vs Does not halt (or diverges): Given a partial computable function (or a Turing machine) φ, and some input x, we will write $\varphi(x) \uparrow$ to mean that $\varphi(x)$ does no halt on input x, and we will write $\varphi(x) \downarrow$ to mean that φ halts on input x.

To see an example, we can express the Kleene halting set K as

$$K = \{e \in \mathbb{N} \colon \varphi_e(e) \downarrow\}.$$

Exercise 31 **(The Padding Lemma)**

Prove that every partial computable function has infinitely many indices. That is, for any partial computable function φ, the following set is infinite

$$\{e \in \mathbb{N} \colon \varphi_e = \varphi\}.$$

Hint: You can write many programs to compute the exact same function.

[1] a word on Francois Viete

Exercise 32 **(The Parameter Theorem, also known as the s-m-n Theorem)**

Prove that there exists an injective computable function $s_n^m \colon \mathbb{N}^{m+1} \to \mathbb{N}$ such that, for all $x, y_1, \ldots, y_m, z_1, \ldots, z_n \in \mathbb{N}$:

$$\varphi_x(y_1, \ldots, y_m, z_1, \ldots, z_n) = \varphi_{s(x,y_1,\ldots,y_m)}(z_1, \ldots, z_n).$$

Indexing recognizable sets: Since, by definition, every recognizable set is the domain of some partial computable function, we can give each recognizable set an index, which is the Gödel number of the partial computable function whose domain is that set. We write W_e to mean the domain of φ_e, that is $W_e = \{x \colon \varphi_e(x) \downarrow\}$. Therefore, the following is the collection of all recognizable sets: $\{W_e \colon e \in \mathbb{N}\}$.

Here is another way to express Kleene's halting set: $K = \{e \in \mathbb{N} \colon e \in W_e\}$.

Oracle Notation: The notation we have just discussed can all be adapted to fit an oracle. For example, we can have φ_e^A to denote the oracle machine with oracle A that has Gödel number e. We can also have $W_e^A = dom(\varphi_e^A)$. Even Kleene's halting set can be relativized:

$$K^A = \{e \in \mathbb{N} \colon e \in W_e^A\}.$$

We have seen already two important results in exercises 31 and 32. Notice that these results focused on indices of partial computable functions. Sets of indices of partial computable functions are interesting as we will see in this chapter.

Definition 1 **(Index Set)**

A set A of natural numbers is an *index set* if for all $x, y \in \mathbb{N}$:

$$(x \in A \wedge \varphi_x = \varphi_y) \Longrightarrow y \in A.$$

The implication says that: if A contains an index (Gödel number) of some partial computable function, then A must contain all the other indices of that partial computable function.

Example 1 The following are popular examples of index sets:

Tot $= \{e \colon \varphi_e \text{ is total } \}$

Fin $= \{e \colon W_e \text{ is finite } \}$

Inf $= \{e \colon W_e \text{ is infinite } \}$

$K_1 = \{e \colon W_e \text{ is nonempty } \}$

Exercise 33 Prove that the examples above are indeed index sets using the definition.

Exercise 34 If the index set is $\mathbb{N}$ or $\emptyset$, we say that it is *trivial*. In other words, being a trivial index set means that the indices do not specify an interesting collection of partial computable functions. Prove that all the examples above are nontirivial.

As you can see, index sets are sets of natural numbers, which means we can study their decidability, recognizability, etc.

Theorem 2 **(Rice's Theorem)**

The only decidable index sets are the trivial ones.

This is a deep theorem, it says that any interesting property of partial computable functions is impossible to decide. For example, no computer program can tell us for any given e whether φ_e is total or not. None of the index sets given in Example 1 is decidable. Most of them are not even recognizable, they are more unsolvable[2].

To prove Rice's Theorem, one shows that, if A is neither $\mathbb{N}$ nor $\emptyset$, then $K \leq_m A$ or $K \leq_m \overline{A}$ (or both). Can you see why showing this implies the statement in Rice's Theorem?

[2]Soon you will learn about the *Arithmetic Hierarchy*.

Chapter 7

Computable Approximations

Chapter 8

The Arithmetical Hierarchy

The idea of relativization allows for exploring different levels of computability (and consequently of decidability and of recognizability). For example, one can show that all recognizable sets are decidable relative to H.

Definition 1 **(The Jump Operator)**

For a set A of natural, the relativized Kleene halting set K^A is known as the *jump of* A.

Chapter 9

Related Topics

9.1 Hilbert's 10th Problem

9.2 Gödel's Incompleteness Theorems

Part II

Theory of Complexity

On the other hand, complexity theory goes a step further by categorizing solvable problems based on the resources required—time, space, or other measures. This distinction is critical because, in the real world, resources are finite, and understanding the complexity of a problem can guide us in choosing the most efficient algorithms, leading to significant improvements in performance and cost.

Something on circuit complexity

https://math.stackexchange.com/questions/756813/do-there-exist-polynomials-not-computable-in-polynomial-time