

## COMPUTABILITY AND INCOMPLETENESS FACT SHEETS

### Background

*Basic notions:* I will assume that there is a basic notion of a *symbol*, and a *string* of “symbols.” For example, if ‘h’, ‘e’, ‘l’, and ‘o’ are individual symbols, then “hello” is a string of symbols.

In discussing computability, we will be concerned with two types of objects: *functions* taking strings of symbols to strings of symbols, and *sets* of strings of symbols. Here are some examples of functions:

- *Multiplication:* The multiplication function expects as input a string of two decimal numbers, separated by a comma (e.g. “12,17”) and outputs their produce (e.g. “204”)
- *Reversal:* The “reverse” function takes a string of letters (e.g. “hello”) and reverses it (“olleh”)
- *Sort:* This function takes a list of words, separated by commas, and outputs the same list sorted in alphabetical order
- *Traveling salesman problem:* takes as input a list of cities, and distances between them, and returns the shortest trip the visits each city exactly once.

Here are some examples of sets:

- The set of prime numbers.
- The set of strings of symbols whose length is even.
- The set of strings representing formulas that are derivable in (some consistent part of) Frege’s logic.

One can think of a set as a special type of function that, for each input, outputs either “yes” or “no”.

Below, when I refer to a function  $f$ , I am implicitly assuming that there is a finite set of symbols  $\Sigma$  such that  $f$  is a function from strings of symbols in  $\Sigma$  to strings of symbols in  $\Sigma$ . I will also adopt the analogous convention for sets.

Most of the examples above are, intuitively, “computable.” Our goal is to give a *precise* definition of what it means for a set or function to be computable. Then we will consider some examples of interesting sets and functions that are *not* computable.

*Note:* Below I will use the terminology “computable” and “computably enumerable.” The textbook uses the terminology “recursive” and “recursively enumerable” instead. Don’t let this confuse you; the terms “computable” and “recursive” mean the same thing.

## Computability

*Definition.* A *Turing machine* is given by:

- A finite set of *symbols*,  $s_1, \dots, s_m$  (including a “blank” symbol)
- A finite set of *states*,  $q_1, \dots, q_n$  (including a special “start” state)
- A finite set of *instructions*, each of the form

If in state  $q_i$  scanning symbol  $s_j$ , perform act  $A$  and go to state  $q_k$

where  $A$  is either “move right,” “move left,” or “write symbol  $s_l$ .”

The notion of a “computation” of a Turing machine can be described in terms of the data above.

*Definition.* Let  $f$  be a function from strings to strings. Then  $f$  is *computable* (or *recursive*) if there is a Turing machine  $M$  that works as follows: when  $M$  is started with its input head at the beginning of the string  $x$  (on an otherwise blank tape), it eventually halts with its head at the beginning of the string  $f(x)$ .

*Definition.* Let  $S$  be a set of strings. Then  $S$  is *computable* (or *decidable*, or *recursive*) if there is a Turing machine  $M$  that works as follows: when  $M$  is started with its input head at the beginning of the string  $x$ , then

- if  $x$  is in  $S$ , then  $M$  eventually halts, with its head on a special “yes” symbol; and
- if  $x$  is not in  $S$ , then  $M$  eventually halts, with its head on a special “no” symbol.

*The Church-Turing Thesis.* A function is *computable* in the intuitive sense if and only if it is *computable* according to the definition above; and similarly for sets.

*Turing's evidence for the C-T thesis:*

1. a careful analysis of the intuitive notion of “computability”
2. the demonstration that lots of and lots of functions that we would intuitively call “computable” are, in fact, computable by a Turing machine
3. the demonstration that other proposed definitions of computability in fact yield the same class of functions

### Unsolvable problems

*Definition.* Let  $S$  be a set of strings. Then  $S$  is *computably enumerable* (or *semi-decidable* or *recursively enumerable*) if there is a Turing machine  $M$  that works as follows: when  $M$  is started with its input head at the beginning of the string  $x$ , then:

- If  $x$  is in  $S$ , the machine halts with its head on a special “yes” symbol; and
- If  $x$  is not in  $S$ , the machine never halts.

It should be clear that any computable set is computably enumerable. (Why?) The following shows that there are computably enumerable sets that are not computable.

*Definition.* Suppose we fix a reasonable encoding of Turing machines as strings of symbols. Let  $K$  be the set of encodings of pairs  $\langle M, x \rangle$  such that Turing machine  $M$  halts on input  $x$ . The problem of deciding whether or not a string is in  $K$  is known as the *halting problem*; it amounts to answering the question “does machine  $M$  halt on input  $x$ ?”

*Theorem (Turing).*  $K$  is computably enumerable, but not computable.

Turing showed that  $K$  is computably enumerable by first showing that there is a “universal Turing machine,” which can simulate the behavior of any machine that is specified as input. A machine can then decide  $K$  as follows:

on input  $\langle M, x \rangle$ , use the universal Turing machine to simulate the behavior of  $M$  on input  $x$ , and return “yes” if the simulation eventually halts.

Let us sketch a proof that  $K$  is not computable. Suppose otherwise; that is, suppose  $K$  is computed by a machine  $M$ . Then we could build a machine  $N$  that works as follows. On input  $x$ :

1.  $N$  calls  $M$  with the question, “does machine  $x$  halt on input  $x$ ”?
2. If the answer is “yes,”  $N$  goes into an infinite loop. If the answer is “no,”  $N$  halts.

Now ask, does  $N$  halt when you input an encoding of  $N$  itself? First  $N$  calls  $M$  with the question “does  $N$  halt on input  $N$ ?” But if  $M$  says “yes,”  $N$  doesn’t halt; and if  $M$  says “no,”  $N$  halts. This contradicts the assumption that  $M$  decides the question correctly. So there is no such machine  $M$ .

It turns out that undecidable problems can arise in very natural contexts. Here are two more examples:

1. Consider predicate logic with at least one binary relation symbol. The question, “is formula  $F$  provable from the axioms and rules of predicate logic?” is computably enumerable but not computable.
2. “Peano arithmetic” is a simple set of axioms for arithmetic. The question, “is formula  $F$  provable from the axioms of arithmetic?” is computably enumerable but not computable.

## Incompleteness

Consider strings of symbols that represent formulas in predicate logic.

*Definition.* Let  $A$  be a set of axioms.

- $A$  is *complete* if for every sentence  $F$  in the language, either one can prove  $F$  from  $A$ , or one can prove  $\sim F$  from  $A$ .
- $A$  is *consistent* if there is no sentence  $F$  such that one can prove both  $F$  and  $\sim F$  from  $A$ .

Here are some natural questions:

1. Is there a complete, computable set of axioms for mathematics?

2. Given a set of axioms for mathematics, can one prove that the axioms are consistent (using only “minimal” assumptions)?
3. Given a mathematical statement, is there an algorithm that decides whether or not the statement is true, or provable from a given set of axioms?

Hilbert proposed representing mathematics with formal languages and formal proof systems, for the purpose of addressing these questions.

*Gödel's first incompleteness theorem.* There is no complete, consistent, computable set of axioms strong enough to prove a certain collection of basic facts of arithmetic.

*Gödel's second incompleteness theorem.* No consistent, computable set of axioms strong enough to prove a certain collection of basic facts of arithmetic can prove its own consistency.

*Theorem.* There is no algorithm that decides whether or not a statement of arithmetic is true. Given a “reasonable” set of axioms for arithmetic, there is no algorithm that decides whether or not a formula is provable from the axioms.