Notes on Sipser: § 3.3, “The Definition of Algorithm”
CSC 341, “Automata, Formal Languages, and Computational Complexity”
Department of Computer Science · Grinnell College
August 31, 2020

Since the publication in 2013 of the current (third) edition of *Introduction to the Theory of Computation*, the informal use of the word ‘algorithm’ among English speakers has changed significantly. It is now commonly used to refer to the workings of machine-learning systems and other artificial-intelligence applications. These are not algorithms in the computer scientist’s sense of the term, however, because the problems to which they are applied are not (and usually cannot be) stated formally and because the AI applications sometimes produce incorrect answers to instances of those problems or fail to terminate.

In the technical sense, an *algorithm* is an effective step-by-step method for answering any question that is an instance of a well-defined, formally stated problem, usually one that is parameterized, so that the instances of the problem correspond to different values of the parameters. The algorithm receives as inputs the values of the parameters of the problem and delivers as output the (correct) answer to the specified instance.

The term ‘effective’ means that the algorithm always terminates and delivers the answer after a finite amount of computation, and that each step in the algorithm is a well-defined operation that requires a finite amount of computation and can be executed mechanically, without requiring special insight or guesswork on the part of the operator.

Sipser understates the shocked reaction of mathematicians to the discovery that some problems that can be stated formally cannot be solved algorithmically. Nearly all mathematicians of the early twentieth century shared Hilbert’s assumption that it must be possible to find algorithms that solve such problems as whether a polynomial containing two or more variables has an integral root, just because all the terms and logical structures used in the statement of that problem can be defined formally. Kurt Gödel’s 1930 paper “On Formally Undecidable Propositions in *Principia Mathematica* and Related Systems I” was the first indirect adumbration of the idea that there could be such a thing as an algorithmically unsolvable problem in mathematics. Even after the 1936 papers by Church and Turing, many mathematicians clung to the idea that perhaps algorithmically unsolvable problems were all pathological in some way, or that they came up only in peripheral fields of mathematics such as logic, set theory, and automata theory. The importance of Matijasević’s discovery, published in 1970, about the problem of the integer roots of multivariable polynomials is that it implied that algorithmic insolubility and undecidability could arise in any branch of mathematics.

Some readers may be skeptical about the possibility of encoding arbitrary objects as strings. Sipser curtly dismisses such concerns:

The encoding itself can be done in many reasonable ways. It doesn’t matter which one we pick because a Turing machine can always translate one such encoding into another.

If you’re struggling a little with this idea of universal encoding, consider that every datum in a computer program, regardless of its type, ultimately has to be stored in the
computer as a linear sequence of byte values. By making every integer from 0 through 255 an element of the Turing machine’s alphabet $\Sigma$, we could “encode” the datum just by transcribing the linear sequence of bytes onto the tape. There is a complication for data containing pointers—the transcriber would have to dereference such pointers and add the datum at the other end to the transcription. There would also have to be a special convention for circular data structures, so that the transcriber could avoid dereferencing any pointer to a datum that has already been transcribed once. But all such data structures are finite, and the transcriber will eventually reach, label, and transcribe all of their components.

Alternatively, if you prefer, we could write out a formal mathematical description of any object and encode it just by listing the Unicode codepoints of the mathematical symbols used in that description—the alphabet in a Turing machine could include all Unicode codepoints as a subset. Or we could write out an algorithm for constructing the datum in some programming language such as Scheme and use that Scheme code as the input.

In short, all Sipser is really assuming here is that, if an object $O$ is sufficiently well-defined that we can write a mathematical description of it or an algorithm that generates it, then we can encode into that mathematical description or express that algorithm using a finite alphabet of symbols. That’s not so implausible.

---

You’ll probably want to read and think carefully about Problem 3.22, which seems at first to be asking the reader either to create a Turing machine that can answer an open-ended empirical question, one that probably cannot even be stated formally, or else to prove that no such Turing machine exists. The problem is actually asking for something subtly different and much easier to provide. Hint: Consider using a disjunctive syllogism to structure your solution.