Computability

Author(s)
Silvio Peroni – silvio.peroni@unibo.it – https://orcid.org/0000-0003-0530-4305
Digital Humanities Advanced Research Centre (DHARC), Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Keywords
Alan Turing; Computability; Halting problem; Turing machine

Abstract

This chapter introduces the notion of *computability* and the *computational cost* of algorithms. The historic hero presented in these notes is Alan Turing, considered the father of Theoretical Computer Science and Artificial Intelligence. His work on a particular model of computation, known as the *Turing machine*, had been the primary tool for highlighting the possibilities and the limits of automatic computation and, more in general, of the modern electronic computer.

Historic hero: Alan Turing

Alan Mathison Turing (shown in Figure 1) was a computer scientist. His works spanned several disciplines, including mathematics, logic, philosophy, and biology – which is why people have referred to him as a natural philosopher [Dodig-Crnokovic, 2013]. He is considered the father of Theoretical Computer Science and Artificial Intelligence, due to its frontier contributions that provided his theoretical machine named after him [Turing, 1937][1]. Besides, his studies on the relation between electronic computers and intelligence [Turing, 1950] brought him to define the thought experiment known as the Turing test.

---

[1] The Turing machine has been used for modelling plenty of situations in several domains, such as cellular automata [Wolfram, 1983] [Wolfram, 2002] in Applied Physics.

He was one of the key figures behind the decryption of Enigma, the cypher machine used by Nazi Germany for protecting communications[2]. Also, his studies do not focus only on Computer Science topics. In fact, they include essential works in Biology. In particular, one where he described the way patterns in nature (e.g. stripes, spots and spirals) may arise spontaneously out of a uniform state [Turing, 1952].

Figure 1. Picture of Alan Turing taken in 1927. Source: https://en.wikipedia.org/wiki/File:Alan_Turing_Aged_16.jpg.

---

[2] A story that has been recently portrayed as a movie by Morten Tyldum's *The Imitation Game*.

The Turing machine

In 1936, Turing developed his machine to answer an important issue related to Hilbert's *decision problem* which asks about the possibility of developing an algorithm for deciding if a first-order logic formula is universally valid or not. The problem was also analysed at the same time by Alonzo Church, by addressing it from a totally different (but pragmatically equivalent) perspective compared with Turing's approach. The machine proposed by Turing was only theoretical. Indeed, he did not build it physically. Recently, several people provided physical prototypes of Turing's idea, such as the one shown in Figure 2.
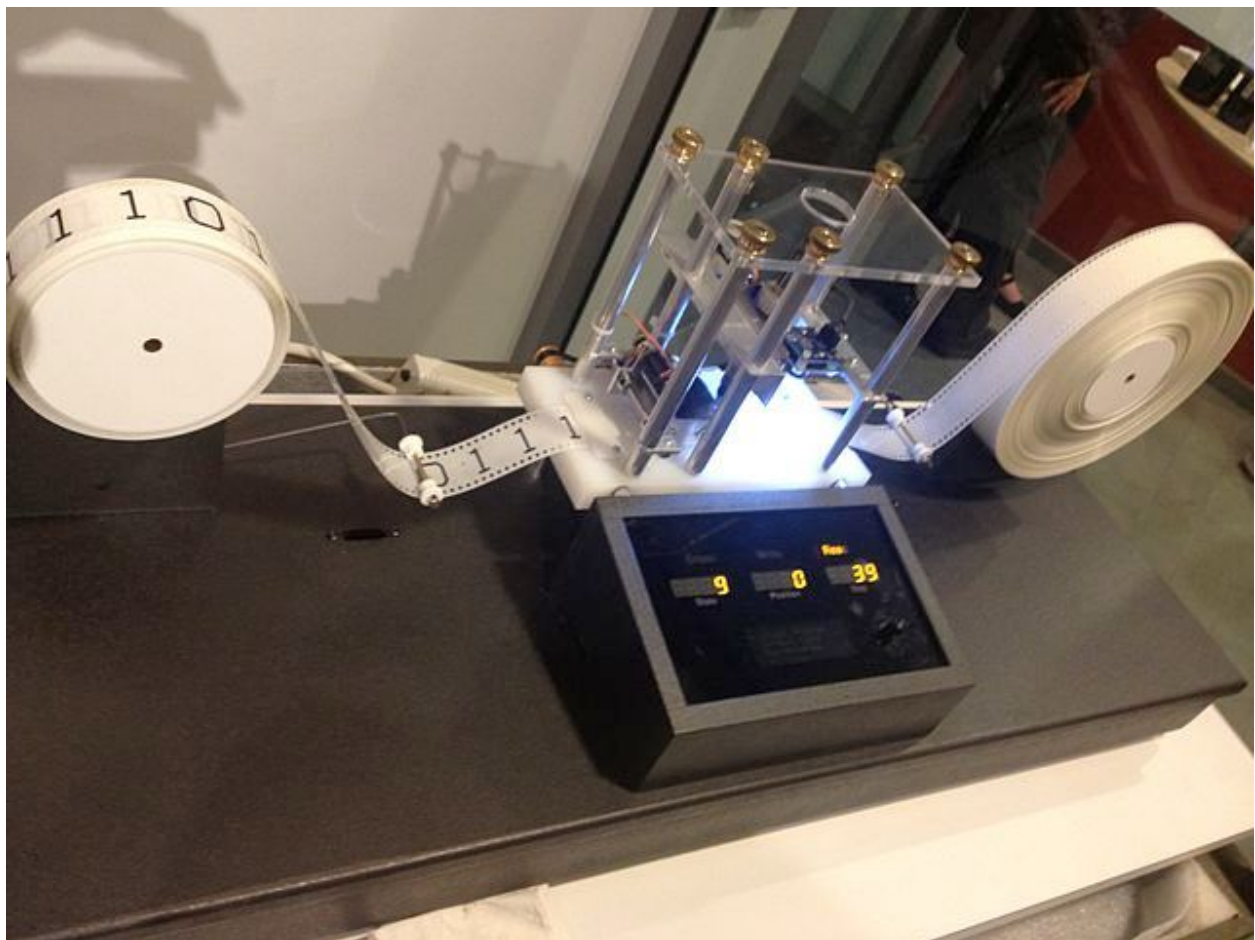

Figure 2. A physical implementation of a Turing machine with a finite tape. Picture by GabrielF, source: https://commons.wikimedia.org/wiki/File:Model_of_a_Turing_machine.jpg.

Broadly speaking, the Turing machine **can be used to simulate any algorithm** by means of a quite simple set of tools. In fact, it is composed of an infinite memory tape containing cells. Each cell can contain a symbol (i.e. either *0* or *1*, where *0* is the blank symbol, assigned to all the cells in advance) that can be read and written by the head of the machine. The state of the machine

at a specific time is recorded. The machine specifies the possible actions to perform in a finite table of instructions. Each instruction in the table says what to do: write a new symbol, move the head either left or right, go to a new state. The machine selects a particular instruction to execute according to the current state and the symbol currently under the head. An initial state and zero or more final states are provided, to define where to start the process, and when to finish it.

For instance, in Table 1, there is a representation of a table of instructions for a simple Turing machine, where: *A* is the initial state, and there are no final states. Each row in the table represents a particular instruction. For instance, the first row says that being in *A*, if the head reads *0* or *1* on the tape, then *1* is written down, the head is moved one cell to the right, and the new state of the machine becomes *B*.

Current state	Tape symbol	Write symbol	Move head	Next state
A	0 or 1	1	right	B
B	0 or 1	0	right	A

Table 1. A table of instructions of a very simple Turing machine, having initial state *A*, with no final states.
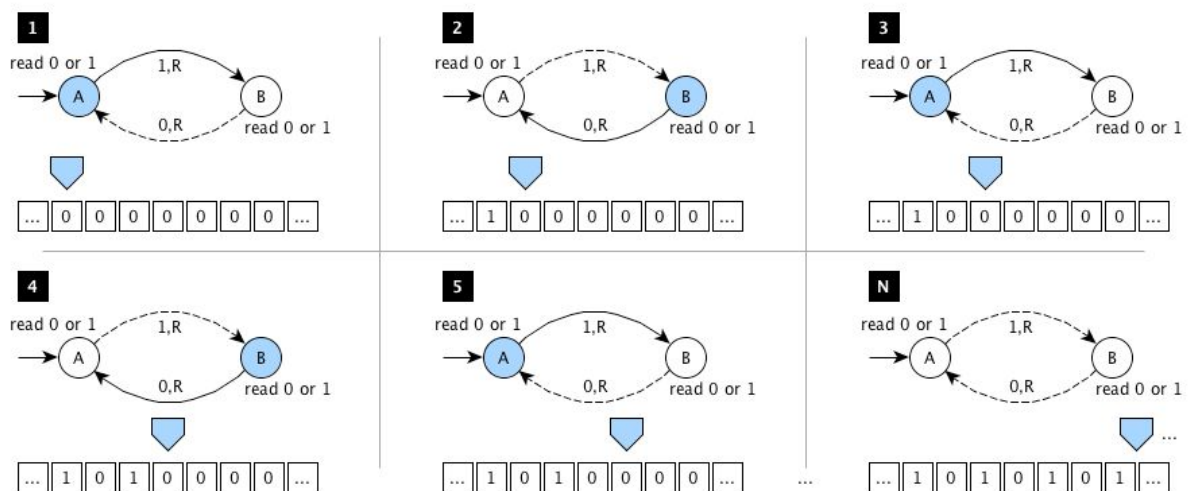

Figure 3. A graphical representation of the execution of the Turing machine implementing the rules introduced in Table 1.In the various figures, the blue polygon represents the head of the machine, positioned in a specific cell of the tape. The blue circle represents the current state, while the solid arrow depicts the next state, reached once the machine writes the symbol in the label of the solid arrow in the cell pointed by the head. Finally, the machine moves the head in the direction indicated on the label (where *R* stands for *right*).

Also, we can represent the table of instructions of a Turing machine graphically. We can use labelled circles for representing states. Also, we can use arrows pointing to the next states when a particular symbol is read on the tape. The machine writes the symbol indicated in the label of an arrow when such arrow is followed. Similarly, the head of the machine is moved according to the direction (i.e. left or right) indicated in the label of the arrow. For instance, in Figure 3, it is shown the execution of the Turing machine related to the table of instructions introduced in Table 1. In particular, this Turing machine has the characteristic of running forever – it will never stop its execution – since it writes several *1s* separated by *0s* indefinitely. Practically speaking, this Turing machine demonstrates that it is possible to **develop algorithms that run forever** without ever ending their execution.

While the Turing machine is quite a simple tool, it enables one to model *computation* in the broad sense. While Turing has not proposed it as a sketch for the development of electronic computers, its theoretical properties apply also to real computing machines. In particular, an electronic computer can compute anything that a Turing machine can compute. This property has been used to prove the intrinsic limitations on the power of mechanical computation.

People have developed several tools to emulate a Turing machine. The Turing Machine Visualization is one of such tools, mainly designed for academic purposes. It is a simple web application that allows one to define all the components of a Turing machine through a straightforward language. Once defined the initial state, initialised the tape with *0*s, and defined the table of instructions, one can watch the way the machine runs. In particular, each step of the execution is shown graphically.

The various variables can be specified by means of the following template:

```
blank: '0'
start state: <start state>
table:
  <state>:
     <tape symbol>: { write: <symbol>, <R or L move>: <next state> }

  <end state>:
```

Where `blank` says how to initialise the tape, the `table` is composed by one or more `<state>`s, one of which is used as `<start-state>`, and one or more (optional) `<end state>`s can be specified as well – they is recognisable since no operations have been defined on them. Following this template, the Turing machine described in Table 1 is defined as follows:

```
blank: '0'
start state: A
table:
  A:
```

```
    0: { write: 1, R: B}
    1: { write: 1, R: B }
  B:
    0: { write: 0, R: A }
    1: { write: 0, R: A }
```

Figure 4 shows the Turing machine implemented by these instructions. Three distinct areas realise the whole visualisation of the machine. A text box contains the rules written according to the template mentioned above. A graph represents the table of instructions of the machine. Finally, a sequence of cells represents the tape, where a yellow box highlights the head of the machine.
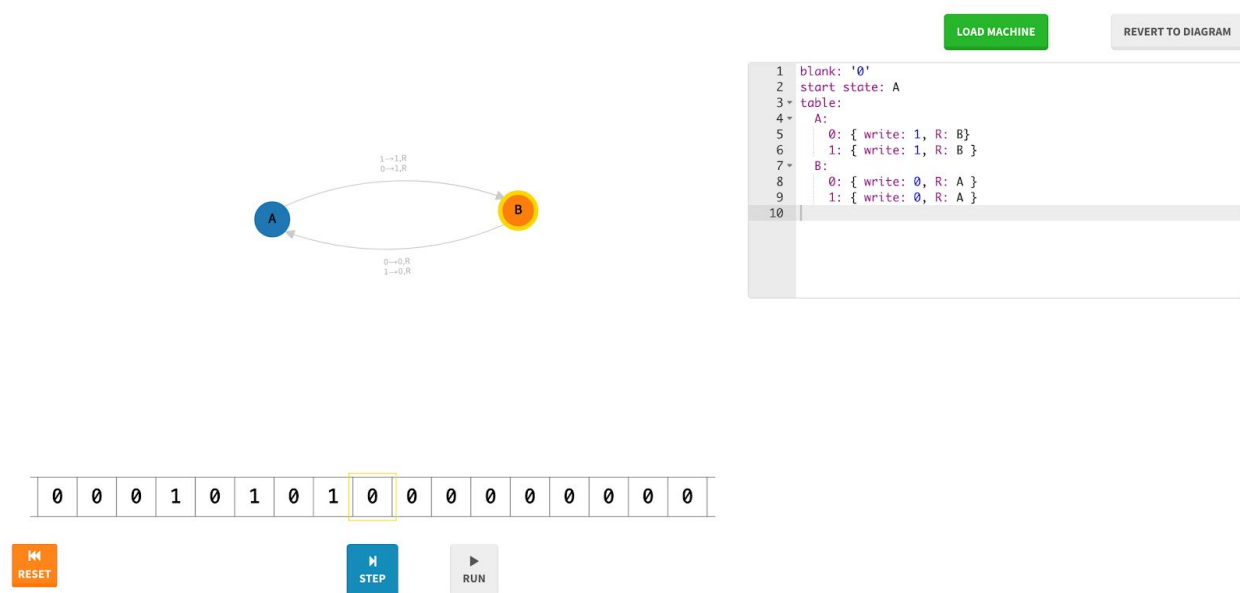

Figure 4. A screenshot of the Web application Turing Machine Visualization.

The computational cost of an algorithm

In the previous lecture, we have defined what an algorithm and the relation that exists between algorithms and computers is. In the previous section, we have already seen how a simple machine, which implements an algorithm using a specific language, can compute indefinitely. Other machines could compute a result in a reasonable finite time. And, even, algorithms can spend an exaggerated time (also if still limited) to return a result. It can be useful to know how much time indicatively an algorithm needs to return a result.

This issue is the core topic of one of the essential branches of the *theory of computation*, i.e. the *computational complexity theory*. The research in this field aims at classifying *computational problems*. A computational problem is a problem that can be solved algorithmically by a

computer. Each computational problem, thus, can be classified according to a hierarchy of categories. Each category expresses the difficulty in solving such a problem.

An important subfield of computational complexity theory is the *analysis of algorithms*. Analysing an algorithm means to understand the amount of time, storage and other resources that are needed to execute such algorithm. In particular, usually, this analysis focuses on finding a particular mathematical function that relates the input of an algorithm with the number of instructions that are run to return the final result from that input. The smaller such function is, the more efficient the algorithm will be.

It is worth mentioning that the measure provided by such function is not precise since it is only an upper bound of the actual performance. However, it is enough for giving an indicative idea of the amount of time needed for executing a particular algorithm on a specific input.

We do not want to introduce all the theoretical principles and the formal mathematical tools for addressing such analysis since it is out of the scope of the course. The message to reinforce is that the strategy used to develop an algorithm affects, positively or negatively, its efficiency. It is possible to create two different algorithms addressing the same computational problem that take two drastic different times for returning the result.

Can we compute everything?

The main question one could ask after reading all the material introduced in the previous sections would be: can we use algorithms for computing whatever we want? In other words: there exists a limitation on what we can compute? Or, even: is it possible to define a computational problem that cannot be solved by any algorithm?

Usually, computer scientists and mathematicians use a *reductio ad absurdum* approach to demonstrate that something cannot exist. This approach aims at coming to a paradoxical and self-contradictory situation, such as the fact that the existence of an algorithm contradicts its existence itself. This kind of argument seeks to establish a contention by deriving an absurdity from its denial, thus arguing that a thesis must be accepted because its rejection would be untenable [Rescher, 2017], and, eventually, generates paradoxes.

Paradoxes have been largely used in Logic in the past. While they are funny stories to tell for teaching, they are also powerful tools for showing limits or constraints of a particular formal aspect of a field or situation. For instance, one of the most famous paradoxes in mathematics is the Russell paradox, discovered by Bertrand Russell in 1901. It was one of the most important discoveries of the beginning of the twentieth century. It has proved that the current set theory proposed by Georg Cantor, and used as the foundation for Gottlob Frege's work on the definition of the fundamental laws of arithmetic, led to a contradiction. Thus, it invalidated the set

theory and the work done by Frege – that was in print when Russell communicated his discovery to him. A variation to that paradox could be formulated as follows.

Librarian paradox: In the Library of Babel, there are people of two different kinds. The first kind of people – named *no-needed* – are those who look for a book themselves. The other type of people – named *help-needed* – are those who do not look for a book themselves, and thus they need help doing it. One of the people in the library is the librarian. The librarian looks for books for all those, and **those only**, who do not look for books themselves (i.e. the *help-needed* people). The question is: who looks for books to the librarian?

Resolution: On the one hand, if the librarian is a *no-needed* person (who looks for a book herself) then the premise that the librarian should look for books only for *help-needed* people would not be valid anymore. Thus, if she is a *no-needed* person, she is an *help-needed* person, which is a contradiction. On the other hand, if the librarian is an *help-needed* person – and, as such, she is not able to look for books herself – she should be helped by the librarian, who is herself! Therefore, if she is an *help-needed* person, she is also a *no-needed* person – which is another contradiction.
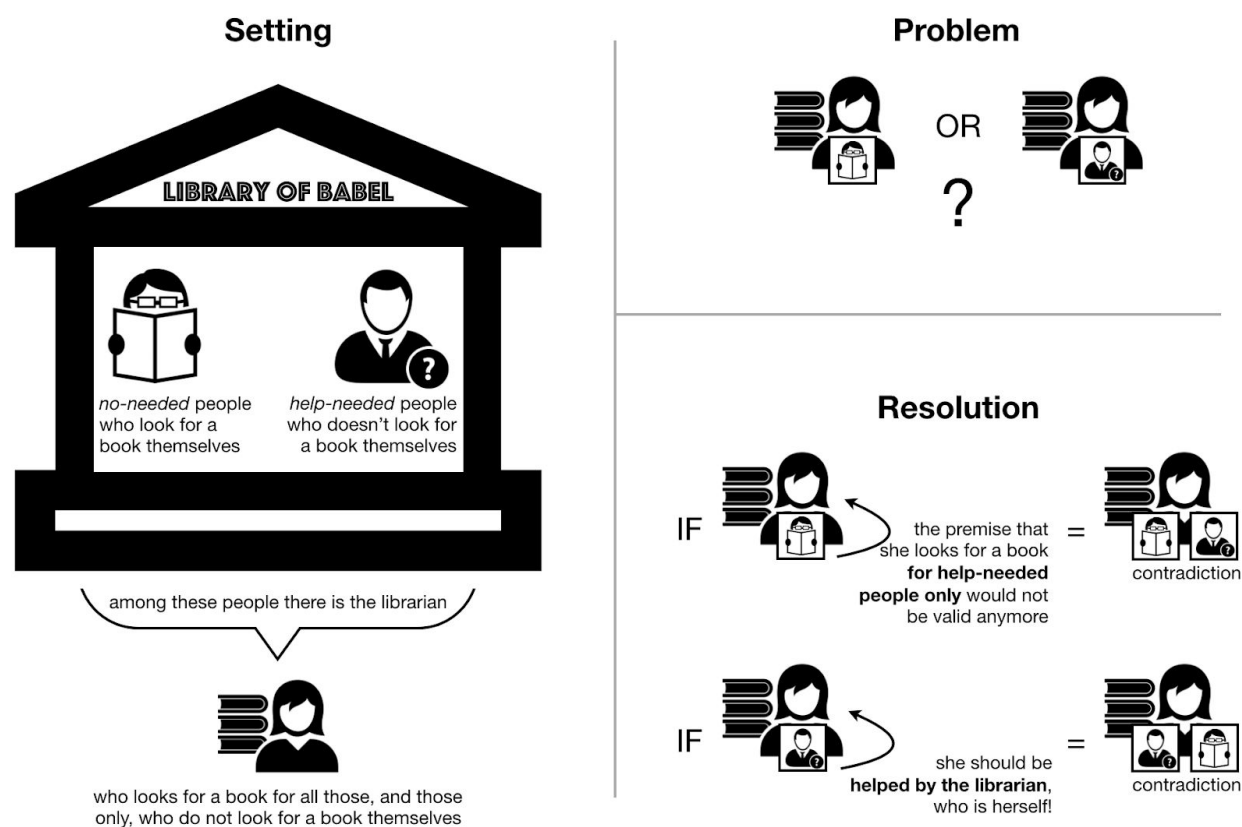

Figure 5. A graphical representation of the librarian paradox, which is a puzzle derived from Russell's paradox.

One of the most attractive problems that computer scientists studied in the past was part of the 23 open mathematical problems that David Hilbert proposed in 1900. It is known as the *halting problem*. This problem was meant to prove whether a particular algorithm will terminate its execution at some point or it will run forever. In the previous lecture, we have developed our first algorithm. We have defined it in a way that allows it to return always a value as an outcome – which confirms that we can create algorithms that terminate. Besides, as demonstrated in Section "The Turing machine", we have shown also an algorithm (implemented by the Turing machine summarised in Figure 3) that runs indefinitely. Thus, having an approach that allows us to discover systematically whether an algorithm will terminate or will not is a great tool to have. Indeed, it would enable the identification of computationally-ill algorithms.

Alan Turing created his machine for answering such a question: to prove if we can develop a Turing machine (i.e. an algorithm) which can decide whether another machine will terminate its execution or will not. An approximation of the solution provided by Turing is introduced as follows. It uses a *reductio ad absurdum* argument, which is very close to the one introduced in Figure 5 for resolving the librarian paradox.

Suppose we have the algorithm "does it halt" as shown in Figure 6, which returns *yes* if the execution of a particular input algorithm terminates, while it returns *no* otherwise. This algorithm is just hypothetical. We are supposing that we can develop it in some way, without providing it in any particular programming language.
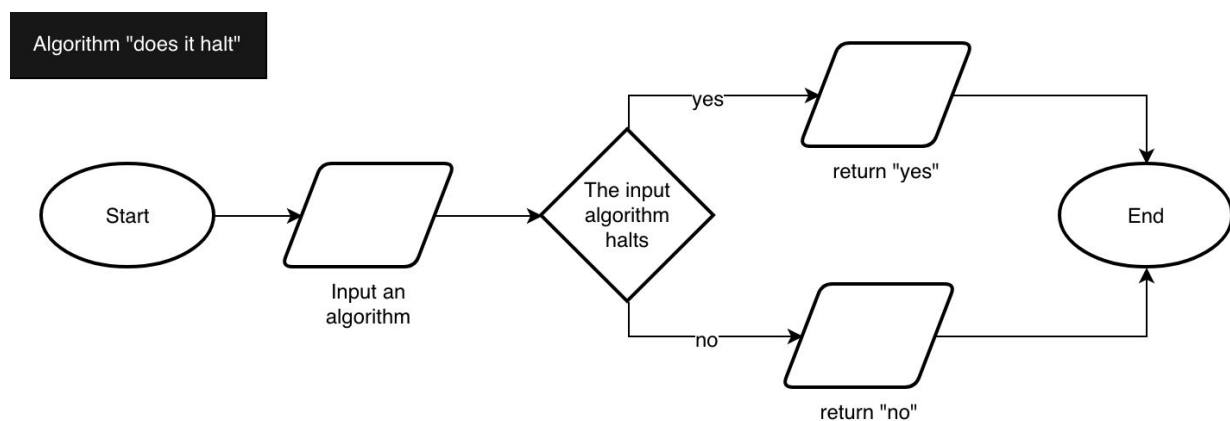

Figure 6. The flowchart of the "does it halt" algorithm, that returns "yes" if the input algorithm halts, and returns "no" otherwise.

Then we reuse the "does it halt" algorithm for developing a new algorithm, shown in the flowchart in Figure 7. In particular, this new algorithm takes another algorithm as input and, if the input algorithm stops, then it runs forever. Otherwise, if the input algorithm does not terminate, then it stops. Please note that we know how to implement the various steps of this new algorithm. In fact, checking whether the input algorithm can terminate or not is actually provided by the algorithm "does it halt" introduced in Figure 5. The "Run forever" process

operation is implementable by a machine, since we have already developed a Turing machine (presented in Section "The Turing machine") that does so.

Now, the question is: what happens if we try to execute the algorithm described in Figure 7 by passing itself as input? We have two possible situations:

- If the algorithm "does it halt" says that our algorithm depicted in Figure 7 **stops**, then our algorithm **runs forever**;
- if the algorithm "does it halt" says that our algorithm depicted in Figure 7 **does not stop**, then our algorithm **stops**.

Hence, whatever is the behaviour of the algorithm introduced in Figure 7, it always generates a contradiction. As a consequence, the main algorithm used in the decision widget, i.e. the algorithm "does it halt", cannot be developed. Thus, the answer to the halting problem mentioned before is that **the algorithm that checks if another one stops cannot exist**.
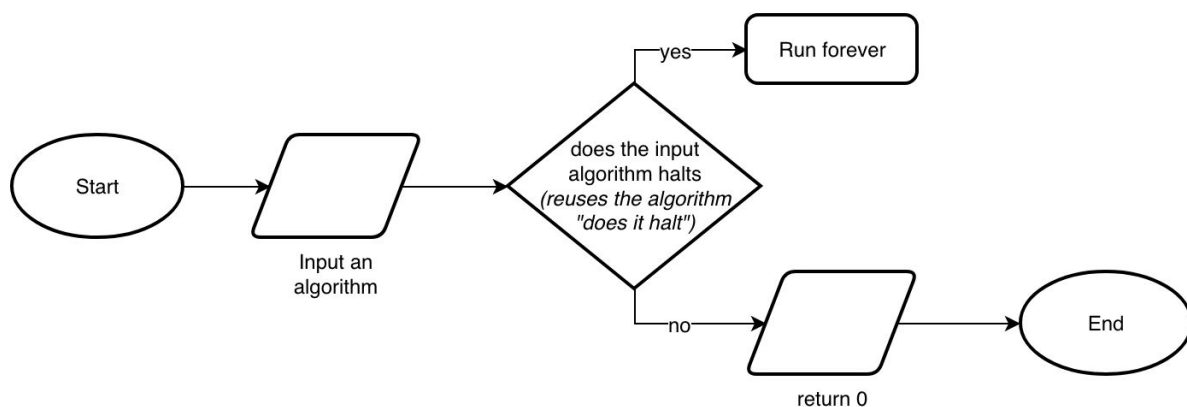

Figure 7. The flowchart of an algorithm that runs forever if the execution of another algorithm specified as input (and checked by using the algorithm presented in Figure 6) stops, and it stops otherwise. Please note that the process step "Run forever" of the flowchart algorithm can be easily developed. In fact, in Section "The Turing machine", we have shown a simple Turing machine that implements such behaviour.

This result had a disruptive effect on the perception of computational abilities at large. In practice, Turing's machines and their analyses posed clear limits to what we can compute. They enabled him to explicitly state that there are specific computational problems, such as the halting problem mentioned in this section, that cannot be solved.

Exercises

1. Write the table of instructions of a Turing machine with four states – A (initial state), B, C, and D (final state) – such that, once reached the final state, only the cells immediately on

the left and on the right of the initial position of the head of the machine will have the value *1* specified. The final state must not have any instruction specified in the table.

2. Consider an algorithm that takes as input a 0-1 sequence of exactly five symbols, and returns a *1* if the sequence contains at least three consecutive *1*s, and returns a *0* otherwise. Implement the algorithm with a Turing machine, where the cell correspondent to the starting position of the head is where the final result must be stored. Also, the five cells following the starting position of the head are initialised with the 0-1 sequence of five symbols used as input of the algorithm.

Acknowledgements

References

Dodig-Crnkovic G. (2013). Alan Turing's Legacy: Info-computational Philosophy of Nature. In Computing Nature: 115-123. Springer. DOI: https://doi.org/10.1007/978-3-642-37225-4_6, also available at https://arxiv.org/ftp/arxiv/papers/1207/1207.1033.pdf

Rescher, N. (2017). Reductio ad Absurdum. Internet Encyclopedia of Philosophy. http://www.iep.utm.edu/reductio/ (last visited 17 October 2019)

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 2 (42): 230-265. DOI: https://doi.org/10.1112/plms/s2-42.1.230

Turing, A. M. (1950). Computing Machinery and Intelligence. Mind, LIX (236): 433-460. DOI: https://doi.org/10.1093/mind/LIX.236.433

Turing, A. M. (1952). The Chemical Basis of Morphogenesis. Philosophical Transactions of the Royal Society of London B, 237(641): 37-72. DOI: https://doi.org/10.1098/rstb.1952.0012, also available at http://www.dna.caltech.edu/courses/cs191/paperscs191/turing.pdf

Wolfram, S. (1983). Statistical mechanics of cellular automata. Review of Modern Physics, 55 (601): 601-644. DOI: https://doi.org/10.1103/RevModPhys.55.601

Wolfram, S. (2002). Two Dimensions and Beyond. In A New Kind of Science: 169-221. Wolfram Media. ISBN: 1579550088, also available at https://www.wolframscience.com/nks/p169--introduction/