Computability

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Keywords
Alan Turing; Computability; Halting problem; Turing machine

Abstract

These lecture notes introduce the notion of *computability* and the *computational cost* of algorithms. The historic hero presented in these notes is Alan Turing, considered the father of the Theoretical Computer Science and of the Artificial Intelligence. His work on a particular model of computation, known as the *Turing machine*, had been the main 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, even if his works spanned several disciplines including mathematics, logic, philosophy, and biology. He is considered the father of the Theoretical Computer Science and of the Artificial Intelligence, due to its frontier contributions that provided his theoretical machine [Turing, 1937], that have been named after him, and his studies on the relation between electronic computers and intelligence [Turing, 1950], which included the thought experiment known as Turing test.

He was also one of the key figures behind the decryption of Enigma, the cypher machine used by Nazi Germany for protecting communications – story that has been recently portrayed as a movie by Morten Tyldum's *The Imitation Game*. In addition, his studies do not focus only on Computer Science topics, but they also include important 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 an uniform state [Turing, 1952].

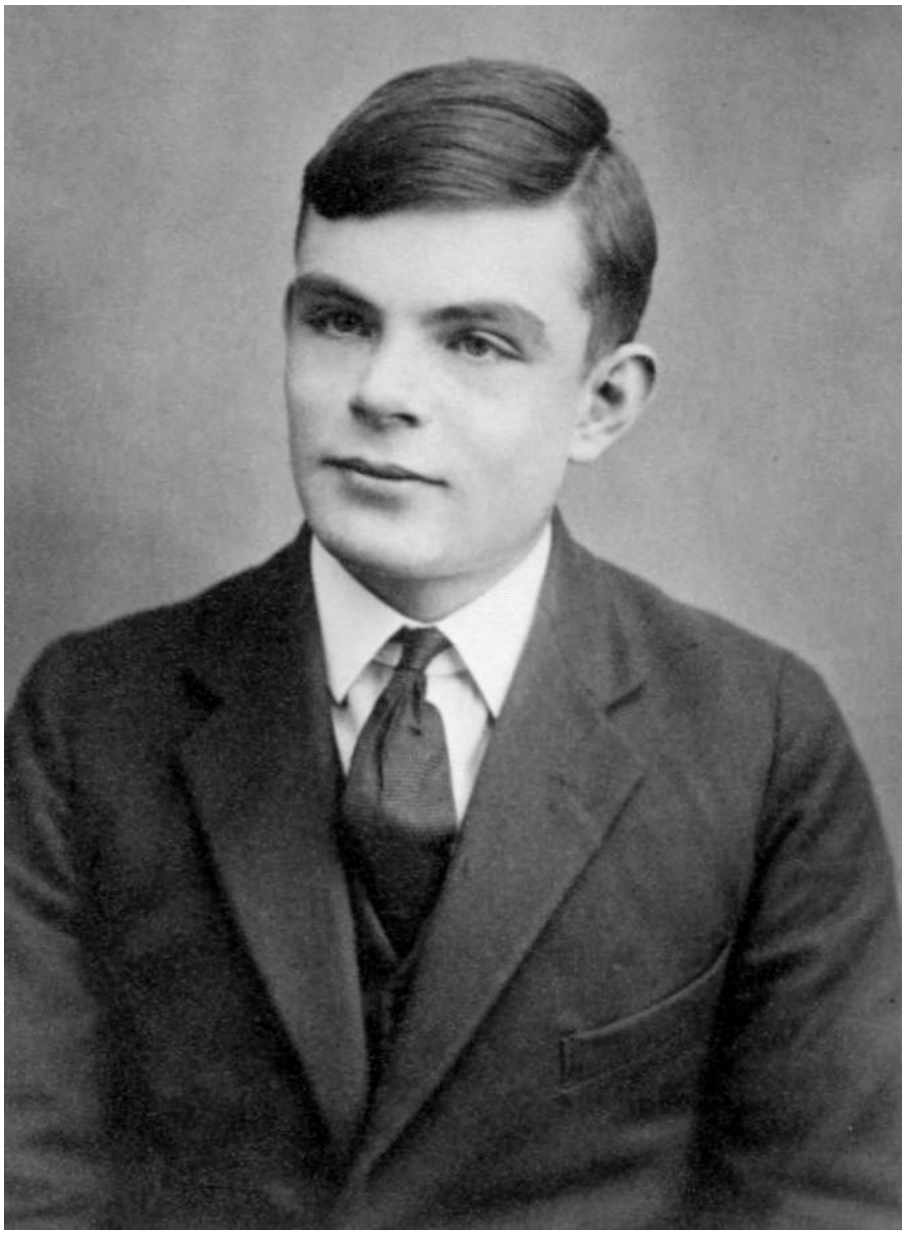

Figure 1. Picture of Alan Turing taken in 1927. Source: https://en.wikipedia.org/wiki/File:Alan_Turing_Aged_16.jpg.

The Turing machine

In 1936, Turing developed its machine so as to answer a quite important issue related with 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 – problem that 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, which means that he did not build it physically – while, recently, several people have provided physical prototypes of Turing's idea, such as the one shown in Figure 2.

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 used as the blank symbol and it is 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 certain time is recorded as well (that describes the possible operations that can be done at the current stage), and the moves of the machine are defined by using a finite table of instructions, where each instruction says what to do (write a new symbol, move the head either left or right, go to a new state) according to the current state and the symbol currently under the head. In addition, an initial state and zero or more final states are provided, so as to know from where to start the process, and when to finish it.

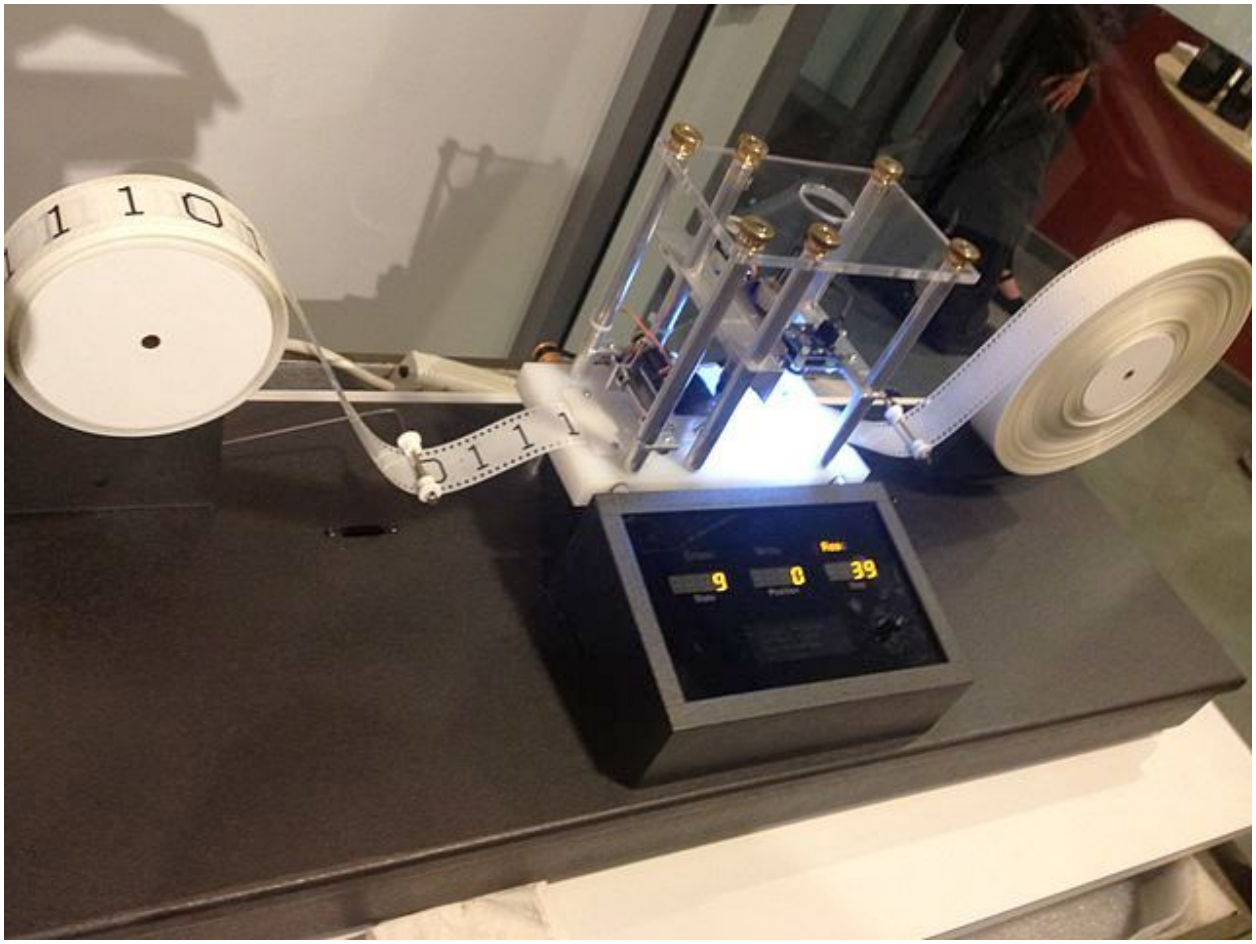

Figure 2. A physical implementation of a Turing machine with a finite tape. Picture by GabrielF, source: https://commons.wikimedia.org/wiki/File:Model_of_a_Turing_machine.jpg.

For instance, in Table 1, there is the representation of a table of instructions for a simple Turing machine, having *A* as initial state, no final states, and using only *0* and *1* as symbols to write on the tape. 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.

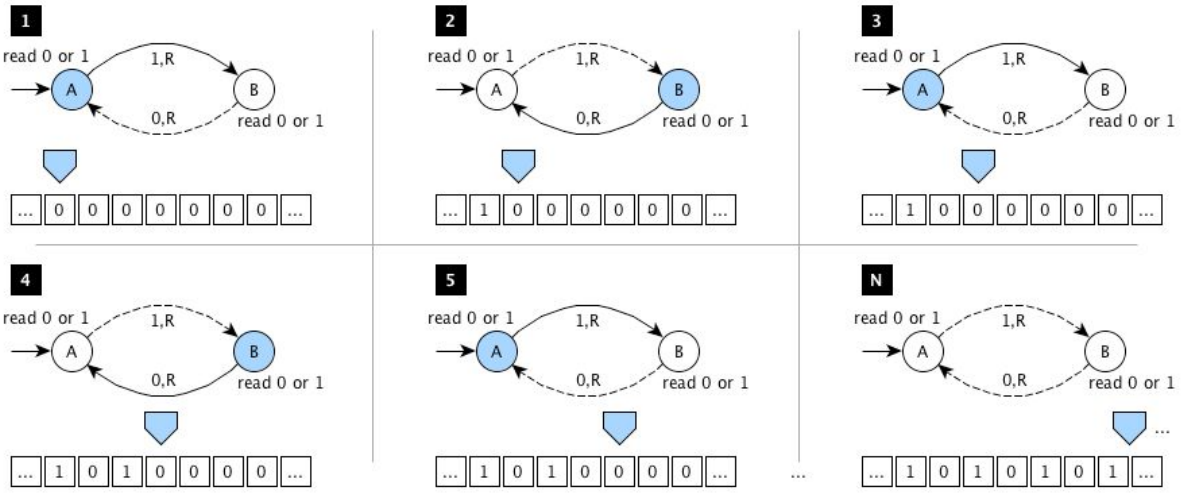

Figure 3. A graphical representation of the execution of the Turing machine implementing the rules introduced in Table 1. In the various figures, the blue polygon represents the head of the machine, which is positioned in a specific cell of the tape. The blue circle represent the current state, while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

The table of instructions of a Turing machine can be also represented graphically by using labelled circles for representing states, and arrows that point to the next state of the current one if a particular symbol is read on the tape. If an arrow is followed – and, thus, the state is changed – the symbol indicated in its label is written down on the table, and the head of the machine is moved according to the direction (i.e. left or right) indicated. 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 a quite simple tool, it enables one to model *computation* in the broad sense. While it has not been proposed by Turing as a sketch for the development of electronic computers, its theoretical properties basically apply also to real computing machines. Thus, anything that can be computed by a Turing machine can be computed as well by an electronic computer, and it has been used to prove the intrinsic limitations on the power of mechanical computation.

There exists several tools that have been developed for simulating a Turing machine. One that can be used for academic purposes is the Turing Machine Visualization. It is a simple web application that allows one to define all the components of a Turing machine by means of a very simple language. Once defined the initial state, the initialisation of the tape (with all "0"s), and the table of instructions, it is possible to see how the machine runs according to an easy diagram that shows the execution of each step.

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 }
```

The Turing machine implemented by these instructions is shown in Figure 4. In particular, the visualisation is split in three parts: a text box containing the rules written according to the aforementioned template, a graph visualisation of the machine, and the tape that will be written by executing the machine.

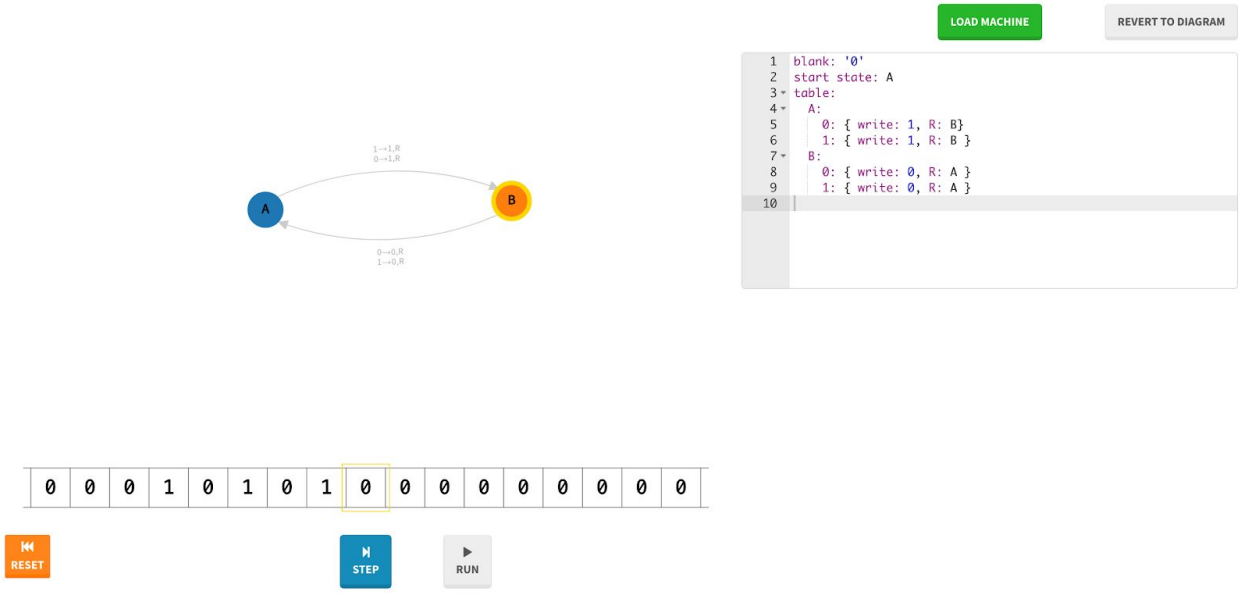

Figure 4. A screenshot of the Web application Turing Machine Visualization.

Computational cost of an algorithm

In the previous lecture, we have defined what is an algorithm and the relation that exists between algorithms and computers. While we have just seen a bunch of widgets available in the flowchart diagram model for defining algorithms, in these lecture notes we have already seen, in Section "The Turing machine", how even a simple machine (simulating an algorithm) can compute indefinitely. Of course, other machines could compute a result in a reasonable finite time, and, finally, algorithms can even spend an exaggerated time (even if still finite) to return a result. Knowing how much time, indicatively, an algorithm needs for returning a result is something that can be useful to know.

This issue is the core topic of one of the most important branches of the *theory of computation*, i.e. the *computational complexity theory*. The research in this field aims at classifying *computational problems* – i.e. problems that can be solved algorithmically by a computer – according to a specific hierarchy of classes that express the difficulty in solving such problems.

An important subfield of the 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 providing an indicative idea of the amount of time needed for executing a particular algorithm on a certain 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. However, the message that we would like to reinforce in this section is that the efficiency of an algorithm is directly derived and guided by the way such algorithm has been developed. It is possible to develop 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?

In the case of the Computer Science domain, as well as of other Mathematical-like sciences, one of the most used approaches to demonstrate that something cannot exist is to come to a paradoxical and self-contradictory situation in which, for instance, the existence of an algorithm contradicts its existence itself, by applying a *reductio ad absurdum* approach. 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, since it has proved that the current set theory proposed by Georg Cantor, and used as foundation for Gottlob Frege's work on the definition of the basic laws of arithmetic, led to a contradiction and, 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 kind of people – named *help-needed* – are those who actually do not look for a book themselves, and

thus they need help for doing it. One of all the people in the library is the librarian – who looks for a book for all those, and **those only**, who do not to look for a book themselves (i.e. the *help-needed* people). The question is, who provides books to the librarian – or, is the librarian a *no-needed* person or an *help-needed* person?

Resolution: If the librarian is a person who looks for a book herself then she is a *no-needed* person. However, she, as the librarian, only helps people that do not look for a book themselves. Hence, she would be also an *help-needed* person, which is self-contradictory: if the librarian is a *no-needed* person, then she is an *help-needed* person. In addition, if the librarian is a person who does not look for a book herself then she is a *help-needed* person. But, in this case, she has to ask herself to look for a book, since she is the librarian, and this generates the second contradiction: if the librarian is a *help-needed* person, then she is a *no-needed* person.

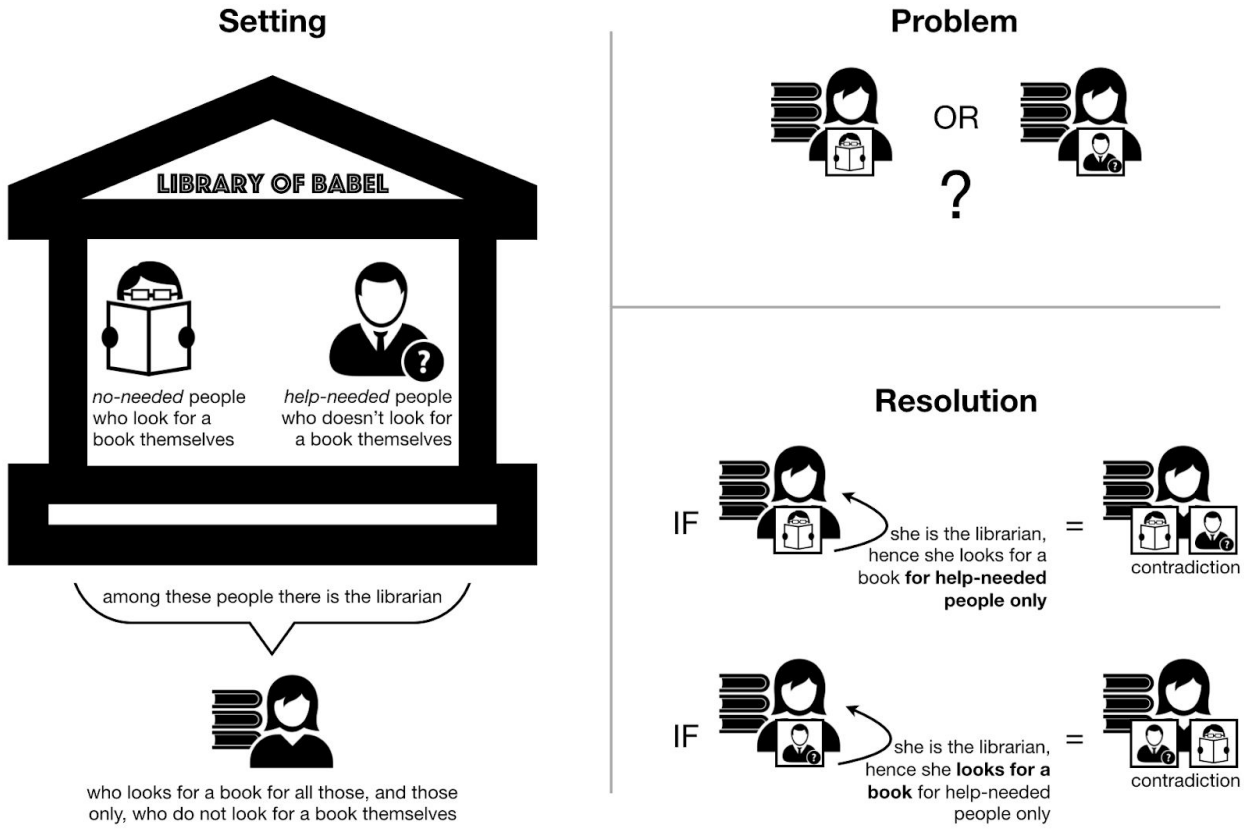

Figure 5. A graphical representation of the librarian paradox, which is a puzzle derived from Russell's paradox.

One of the most attractive problems that were studied in Computer Science in the past was part of the 23 open mathematical problems that David Hilbert proposed in 1900. It is known as *halting problem*, which is the problem of determining if a particular algorithm run with a specific input will terminate its execution at some point or it will run forever. In the previous lecture, we have developed our first algorithm, which has been defined in a way that allows it to return

always a value as outcome – which confirms that we can develop algorithms that terminate. In addition, 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 if an algorithm will terminate or will not is a great utility to have, since it would enable the identification of computationally-ill algorithms.

Alan Turing's development of his machine was done exactly for answering to such question, i.e. if we can develop a Turing machine (i.e. an algorithm) which is able to certainly decide if another machine will terminate its execution or will not. An approximation of the solution that Turing provided is introduced as follows, and it is entirely based on 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 certain input algorithm terminates, while it returns *no* otherwise. This is just a hypothetical algorithm: we are supposing that we can develop it in some way, without providing it in any particular programming language.

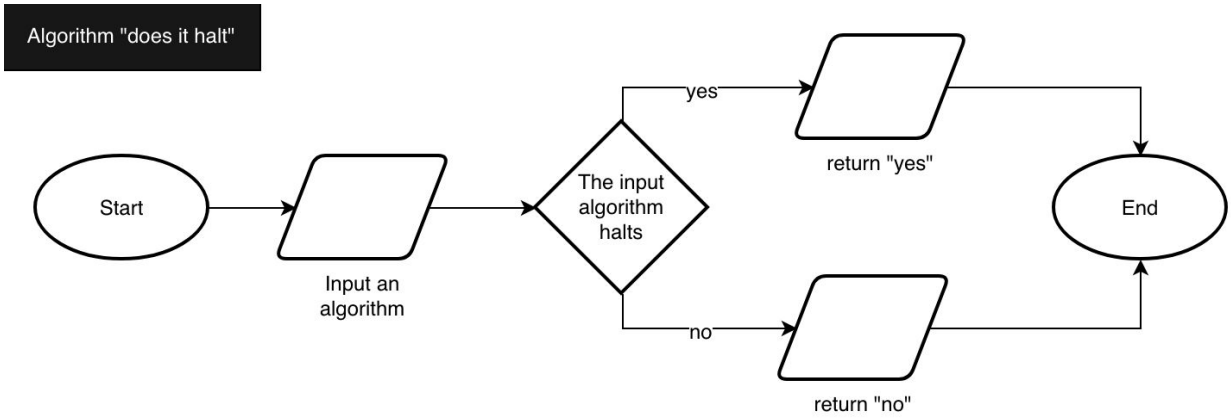

Figure 6. The flowchart of the “does it halt” algorithm, that returns “yes” if the input algorithm halts, and returns “no” otherwise.

Then we reuse the “does it halt” algorithm for developing a new algorithm, that is presented with a 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 actually know how to implement the various step of this new algorithm, since checking whether the input algorithm can terminate or not is actually provided by the algorithm “does it halt” introduced in Figure 5, while the “Run forever” process operation is clearly 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 actually **stops**.

Hence, whatever is the behaviour of the algorithm introduced in Figure 7, it always generates a contradiction. This means that the main algorithm that is 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**.

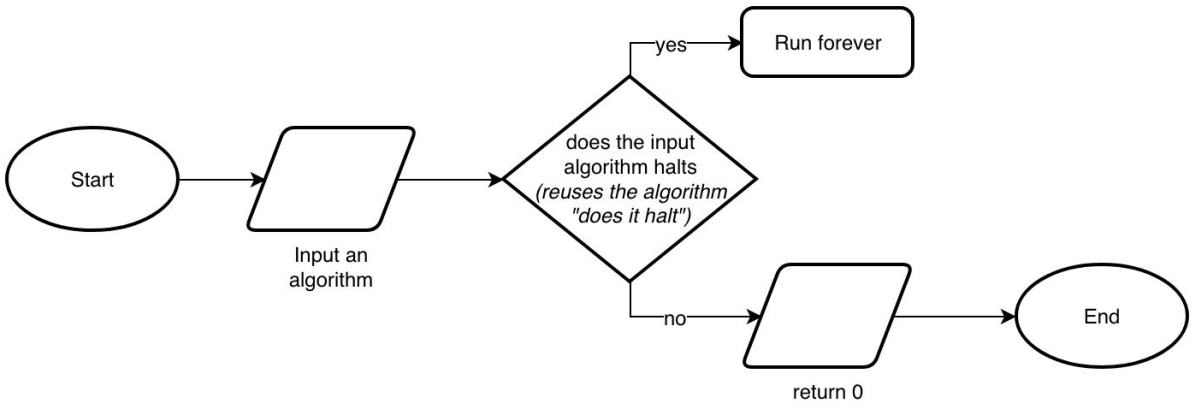

Figure 7. The flowchart of an algorithm that runs forever if the execution of another algorithm specified as input (and checked by using the algorithm presented in Figure 6) stops, and it stops otherwise. Please note that the process step “Run forever” of the flowchart algorithm can be easily developed. 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, and 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 *1* if such sequence contains at least three consecutive *1s*, and returns *0* otherwise. Implement such algorithm with a Turing machine, where the cell correspondent to the starting position of the head is where the final result must be stored, while the following five cells are initialised with the 0-1 sequence of five symbols used as input of the algorithm.

Acknowledgments

I would like to thank a student of the edition of the Computational Thinking and Programming course, i.e. Sebnem Kabadayi, to have suggested the adoption of the Turing Machine Visualization Web application for visualising and running Turing machines.

References

Rescher, N. (2017). Reductio ad Absurdum. Internet Encyclopedia of Philosophy. http://www.iep.utm.edu/reductio/ (last visited 7 November 2017)

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