Lecture 2
Computability

MSc in Physics Engineering - Quantum Computation, L. S. Barbosa, 2021-22

Summary
(1) The halting problem. The uncomputable and the undecidable.
(2) The Church-Turing and the physicalChurch-Turing theses.
(3) An introduction to Turing machines.

---

1 The halting problem

The questions

- Is there an algorithm for each problem?
- What does it mean for a problem to be undecidable, a statement unprovable or a function non computable? Are these notions related?
- Are there any problems/statements/functions of this sort?
- How does the notion of computability depend on each particular set of programming constructs?

Universality implies non halting programs

Since Babbage's Analytical Engine, the hallmark of any programming language is the ability to express an *universal* program, *i.e.* a program that accepts the source code of another program and runs it. Programs can, therefore, run (eventually modified versions of) themselves, which, as it happens in set theory, may lead to paradoxes. The way out from them is to accept that some problems have no algorithm and, thus, cannot be solved.

Let us denote a universal program as $U(P, x) = P(x)$, where P is a program, and x an input. In particular, consider $x = P$, *i.e.* $U(P, P)$ simulates the behaviour of program P when supplied with its own source code as an argument. If, for the sake of simplicity, P returns a Boolean value, one may define a program

$$V(P) = \neg P(P)$$

Giving V its own code as input leads to a paradox:

$$V(V) \;=\; \neg V(V)$$

The way out from this paradox is to admit that program $V(V)$ is *undefined*. Operationally, this means that such a program runs forever and never returns any output. Semantically, one has to admit that total functions are too restrictive to model computational processes: *partiality* is indeed present.

This discussion can be presented as a diagonalization argument, similar to the ones used in the previous lecture to distinguish between different infinite cardinalities. Actually, one can define all universal programs by their output when called with the source code of any other program, as shown in the following table:

$$\begin{array}{llllll}
P_1 & \text{such that} & P_1(P_1) & P_1(P_2) & P_1(P_3) & \cdots \\
P_2 & \text{such that} & P_2(P_1) & P_2(P_2) & P_2(P_3) & \cdots \\
P_3 & \text{such that} & P_3(P_1) & P_3(P_2) & P_3(P_3) & \cdots \\
\cdots & & & & &
\end{array}$$

If such programs always halt each entry in the matrix is a well-defined output. However, let us build a program V taking the diagonal of the matrix and negating each of its cells. I.e. V is defined such that, for all i, $V(P_i) = \neg P_i(P_i)$. Clearly, V is not in the table. On the other hand it must be, *i.e.* $V = P_k$ for some index k: if a universal program exists so does V. To escape the contradiction we are forced to conclude that some outputs $P_i(P_j)$ are not well defined, *i.e.* that some programs never halt on some inputs. In particular, $V(V)$ cannot be well defined.

Universality implies the existence of *non halting* programs. On the other hand, if programs in a given programming language always halt, the language is not expressive enough to interpret itself.

Given that some programs halt and others don't, is there a way (*i.e.* an algorithm) telling which is which? In other words, given a program P and an input x is there a way to decide if P halts on x (or, in another words, is $P(x)$ well defined?) Such is the *halting problem*, as formulated by A. Turing in 1936.

Suppose, then, one can write an algorithm $\mathrm{Halts}(P, x)$ which receives a program P and a value x as input and decides whether it terminates or not. Such a wonderful program could be used to write the following procedure:

Algorithm 1: $\mathrm{DecideHalt}(P)$.

```
1 if $\mathrm{Halts}(P, P)$ then
      go to 1;
  else
      halt;
  end
```

Diagonalization again strikes back when running $\mathrm{DecideHalt}$ on itself. Program

$$\mathrm{DecideHalt}(\mathrm{DecideHalt})$$

halts iff it does not halt. To escape from contradiction, one has to admit that $\mathrm{DecideHalt}$ is undefined and, therefore, the halting problem *undecidable*.

Note that this undecidability is somehow asymmetric: if the answer to $\mathrm{Halts}(P, x)$ is positive, this can be learnt in a finite amount of time by simulating P until it halts. Actually, Halts can be written as

$$\mathsf{Halts}(P, x) \; = \; \exists_t.\, \mathsf{HaltsAfter}(P, x, t)$$

where $\mathsf{HaltsAfter}(P, x, t)$ is true if P halts on input x after t execution steps. This means that Halts is a combination of a decidable property with an existential quantifier. Note that if we keep track of all inputs for which P halts (for example, executing P for all possible inputs and recording the ones on which it halts), the process may be indeed infinite (if the set Y of such inputs, called the set of positive instances of P, is), but every member of Y will be recorded within a finite amount of time. Therefore, we say that program P enumerates Y or that set Y is *recursively enumerable*.

Let us denote by RE the class of problems whose set of positive instances is recursively enumerable. Then, we state without proof the following result (see the course bibliography for a proof):

- Problems within the class RE are the ones expressed as

  $$\mathsf{R}(x) \; = \; \exists_t.\, \mathsf{DecProp}(x, t)$$

  where $\mathsf{DecProp}$ is a *decidable* property.

- The *halting problem* is in RE. Moreover, any other problem in RE can be formulated as an instance of the halting problem. Because of this, we classify the halting problem as RE-*complete*.

The complement of RE (to be denoted as $\overline{\mathsf{RE}}$) is the set of problems whose set of negative instances is recursively enumerable. Not surprisingly, they can be formulated as the logic duals to RE problems. For example,

$$\mathsf{RunsForever}(P, x) \; = \; \forall_t.\, \neg\mathsf{HaltsAfter}(P, x, t)$$

A problem is *decidable* iff both the sets of positive and negative instances are recursively enumerable, *i.e.* it lives in class $\mathsf{RE} \cap \overline{\mathsf{RE}}$.

From undecidable problems and unprovable truths

As discussed in the previous lecture, the quest for formalising the concept of effective computability started around the beginning of the twentieth century with the development of the formalist school of mathematics and Hilbert's programme to find a complete and consistent set of axioms for all mathematics with the prospect of reducing all of mathematics to the formal manipulation of symbols. Indeed David Hilbert believed that all well-defined mathematical problems were decidable.

The formalist program was eventually shattered by Kurt Gödel's incompleteness theorem, which states that no matter how strong a deductive system for number theory you take, it will always be possible to build simple statements that are true but unprovable. This theorem is essentially a statement about *computability*. Actually, Gödel's first (*there is no consistent system of axioms whose theorems can be listed by an effective procedure, i.e. an algorithm, able to prove all truths* *about the arithmetic of the natural numbers*) and second (*no formal system can prove its own consistency*) incompleteness theorems, were the first of a series of results on formal systems, culminating on Turing's theorem that there is no algorithm to solve the halting problem. This means that some problems are indeed out of the reach of any algorithm. Let us discuss a little more the relationship between undecidable problems and unprovable truths.

In logic, a *formal system* is a set of axioms and inference rules from which reasoning develops. A statement that can be deduced through a *finite* reasoning chain starting from the axioms is called a *theorem*. A formal system is

- *(syntactically) complete* if any statement or its negation is provable from the axioms[1]. First-order logic, for example, is not syntactically complete, since there are sentences expressible in its language that can be neither proved nor disproved from the axioms alone. Another example is Euclidean geometry without the parallel postulate, because some statements in the language (namely the parallel postulate itself) can not be proved from the remaining axioms.
- *consistent* if there is no statement such that both the statement and its negation are provable from the axioms; it is said to be *inconsistent* otherwise.

In 1931, Gödel proved that no sufficiently powerful formal system is both consistent and complete. The argument resorts to a self-referential statement which can be interpreted as *this statement cannot be proved*. If this were false, it could be proved, but the system will no longer be consistent. Thus, it must be true, and hence unprovable, showing that there are truths that cannot be proved[2].

We are not interested in Gödel's proof here, but will discuss how the undecidability of the halting problem can lead to the incompleteness theorem. Actually, programs that execute themselves and sentences that talk about themselves are faces of the same phenomenon.

First notice that the set of theorems in a formal system is recursively enumerable: actually the property that a statement is provable can be formulated as

$$\mathsf{Provable}(s) \;=\; \exists_p.\, \mathsf{Proof}(p,s)$$

where $\mathsf{Proof}(p,s)$ is the decidable (why?) property that p is a proof of s.

[1] This should not be confused with *semantic completeness*, which means that the set of axioms proves all the semantic tautologies of the given language. Its dual, called *soundness*, is concerned with the fact that all theorems constitute semantic tautologies. Both notions resorts to some *semantical* model, *i.e.* a class of mathematical objects interpreting, or giving meaning to the symbols and sentences in the formal system. It is in this sense that Gödel proved that first-order logic is (semantically) complete, meaning that if a sentence is true in any semantic model of first-order logic, then it can be proved from the axioms. From a fine grain perspective, completeness and soundness relate a semantic model with a specific deduction system: for example, one can say that e.g. *natural deduction* (a deductive system) *is complete* for the usual semantics of first-order logic.

[2] Gödel's proof is much more precise: actually the sentence above can just be seen as a language paradox and self-reference is very easy to appear in natural language. What he showed was that this kind of self-reference can happen even in formal systems that talk about the integers, by building a sentence that, on one level, claims that an integer with some property does not exist, but that, on another level, can be interpreted as referring to itself, and asserting that no proof of it exists.

Consider a formal system expressible enough to talk about computation, *i.e.* that can express statements such as $\mathsf{Halts}(P, x)$ and to derive each step in a computation from the previous one. The question is whether such a system can prove all statements concerned with the termination of programs. Suppose that we have a system able to prove all true statements of the form $\neg\mathsf{Halts}(P, x)$. If such a system exists the halting problem would be decidable: we could prove that $P(x)$ halts by executing it until it halts (the script of the execution will constitute a proof), and prove that it does not halt by looking for a proof that, by assumption, the system provides.

As the halting problem is undecidable, we are forced to conclude that such a system cannot exist, *i.e.* that for some programs and inputs, the fact that they will never halt is an unprovable truth. As expected, the existence of unprovable truths corresponds to the existence of undecidable problems.

Exercise 1

Suppose that we enrich our formal system adding the statement $\neg\mathsf{Halts}(P, x)$ as a new axiom, thus making it a theorem in a trivial way. Would this enriched system change anything in the argument above?

---

2 The Church-Turing thesis

When claiming, in the previous section, to have proved the existence of an undecidable problem (the halting problem), we swept under the carpet an important detail. Actually, what was shown was that a halting problem exists for each mature programming language, by *mature* meaning a language able to interpret its own programs. To show the general statement, *i.e.* that the halting problem is really undecidable irrespective of the programming language in which it is expressed, entails the need for the definition of a *model of computation* in which all algorithms can be expressed.

Several definitions of computability were proposed along the first decades of the 20th century. In the previous lecture we have already mention some of these ways to capture that a procedure is computable (in other words, it constitutes an algorithm): if

- encoded in a Turing machine (A. Turing),
- expressed in the λ-calculus (A. Church, S. C. Kleene),
- formulated as a partial recursive function (K. Gödel, J. Herbrand).

That all the formalisms above capture precisely the same intuition about what it means to be *effectively computable*, or in other words, that any reasonable computing device can be simulated by a Turing machine is formulated as a conjecture known as the *Church-Turing thesis*. Formally,

> *Church-Turing thesis.* The class of functions computable by a Turing machine corresponds exactly to the class of functions which can be described by an algorithm.

With the advent of quantum computation this conjecture was strengthened to a claim involving the physical processes that can actually be built:

> *Physical Church-Turing thesis.* The class of functions computable in finite time by a physical device can be computed by a Turing machine.

This means that, given enough time, a classical computer can simulate any physical process with arbitrary precision. Note that, while the Church-Turing thesis is a claim about what counts as an algorithm, this stronger version becomes a claim about the physical universe. It actually means that, in principle, given unlimited resources, what can be computed by a classical or a quantum computer is exactly the same. However, as we will discuss in the next lecture, for all practical purposes, what really matters is implicit in the qualifier *unlimited* above. The amount of resources, namely the execution time, required for a particular computation depends on the computational device (e.g. classical or quantum) used. This area of studied is know as *computational complexity*. Before jumping to it, let us characterise in detail the fundamental model of computation we have been mentioned: Turing machines.

3 Turing machines

Definition and examples

A Turing machine consists of three components:

- a *finite-state control*, i.e. a sort of an automata coordinating the operation of the machine;
- a *semi-infinite tape*, that is delimited on the left end by an endmarker $\prec$ and is infinite to the right, acting as the machine memory;
- a *tape-head* which can read/write on the tape, i.e. move left and right over the tape, reading and writing symbols.

Formally, it is defined as a tuple

$$M = (Q, \Gamma, \delta, s, t, r)$$

where Q a finite set of states, Γ is an alphabet with two special symbols $\prec, \curlywedge \in \Gamma$, s, t and r are, respectively, the initial, accepting and rejecting states, and δ a transition relation as explained below.

The input string is of finite length and is initially written on the tape in contiguous tape cells snug up against the left endmarker $\prec$. The infinitely many cells to the right of the input all contain a special blank symbol $\curlywedge$. The machine starts in the start state s with its head scanning the left endmarker. In each step it reads the symbol on the tape under its head. Depending on that symbol and the current state, it writes a new symbol on that tape cell, moves its head either left or right one cell, and enters a new state. The action it takes in each situation is determined by a transition function

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

The meaning of $\delta(p, a) = (q, b, d)$ is as follows: *when in state* p *scanning symbol* a*, write* b *on that tape cell, move the head in direction* d*, and enter state* q. The machine dynamics, as captured by δ is subjected to the following restrictions:

$\forall_{p \in Q} \exists_{q \in Q} \, . \; \delta(p, \prec) = (q, \prec, R)$	never moves off to the left of $\prec$
$\forall_{p \in Q} \forall_{a \in \Gamma} . \; \delta(p, a) = (q, b, -) \;\Rightarrow\; b \neq \prec$	never writes a $\prec$
$\delta(t, -) = (t, -, -)$ and $\delta(r, -) = (r, -, -)$	never leave the accept (reject) state

The *transition function* can be written as a program, i.e. a sequence of lines each of them specifying a possible transition, cf.

$$(p, a, q, b, d)$$

On each machine cycle a suitable program line (matching the current state and the mark on the draft) is picked and executed through the application of the corresponding transition. If a matching line is not found the machine halts the operation.

The Turing machine accepts its input by entering a special accept state t and rejects by entering a special reject state r. On some inputs it may run infinitely without ever accepting or rejecting.

A *configuration* is a tuple in $Q \times \{w \curlywedge^{\omega} \;\mid\; w \in \Sigma^*\} \times \mathbb{N}$ and denotes a global state of the machine. The configuration $\alpha = (p, z, n)$ specifies a current state p of the finite control, current tape contents z, and current position of the read/write head ($n \geq 0$). For example, the initial configuration on input $x \in \Sigma^*$ is $(s, \prec x \curlywedge^{\omega}, 0)$.

The behaviour of a Turing machine is, thus, captured by the following *transition relation* between configurations:

$$(p, z, n) \;\rightarrow\; \begin{cases} (q, z[b/n], n - 1) \;\Leftarrow\; \delta(p, z_n) = (q, b, L) \\ (q, z[b/n], n + 1) \;\Leftarrow\; \delta(p, z_n) = (q, b, R) \end{cases}$$

Example: $\{a^n b^n c^n \mid n \geq 0\}$

- Start in state s and scans to the right over the input string to check that it is of the form $a^* b^* c^*$.
- Does not write anything on the way across (formally, it writes the same symbol it reads).
- When founding the first blank symbol $\curlywedge$, it overwrites it with a right endmarker $\succ$
- Then it scans left, erasing the first c it sees, then the first b it sees, then the first a it sees, until it comes to $\prec$.
- Then scans right, erasing one a, one b, and one c.
- It continues to sweep left and right over the input, erasing one occurrence of each letter in each pass. If on some pass it sees at least one occurrence of one of the letters and no occurrences of another, it rejects. Otherwise, it eventually erases all the letters and makes one pass between $\prec$ and $\succ$ seeing only blanks, at which point it accepts.

Example: $\{ww \mid w \in \{a, b\}^*\}$

- In a first phase, scans out the input to the first blank symbol, counting the number of symbols mod 2 to make sure the input is of even length and rejecting immediately if not.
- It lays down a right endmarker $\succ$, then repeatedly scans back and forth over the input.
- In each pass from right to left, it marks the first unmarked a or b it sees with an overline.
- In each pass from left to right, it marks the first unmarked a or b it sees with an underline.
- It continues this until all symbols are marked. The objective is to find the center of the input string.

$$\begin{array}{l}
\prec\ a\,a\,b\,b\,a\,a\,a\,b\,b\,a\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ a\,a\,b\,b\,a\,a\,a\,b\,b\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,a\,b\,b\,a\,a\,a\,b\,b\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,a\,b\,b\,a\,a\,a\,b\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,\underline{a}\,b\,b\,a\,a\,a\,b\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,\underline{a}\,b\,b\,a\,a\,a\,\overline{b}\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\cdots \\
\prec\ \underline{a}\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\,\overline{a}\,\overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots
\end{array}$$

In a second phase, repeatedly scans left to right over the input.

- In each pass it erases the first symbol it sees marked with underline but remembers that symbol in its finite control.

- It then scans forward until it sees the first symbol marked with overline, checks that that symbol is the same, and erases it.
- If the two symbols are not the same, it rejects. Otherwise, when it has erased all the symbols, it accepts.

$$\Yleft\ \underline{a}\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\,\overline{a}\,\overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \Yup\ \Yup\ \Yup\ \cdots$$

$$\Yleft\ \Yup\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\ \Yup\ \overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \Yup\ \Yup\ \Yup\ \cdots$$

$$\Yleft\ \Yup\ \Yup\ \underline{b}\,\underline{b}\,\underline{a}\ \Yup\ \Yup\,\overline{b}\,\overline{b}\,\overline{a}\ \Yup\ \Yup\ \Yup\ \cdots$$

$$\cdots$$

$$\Yleft\ \Yup\ \Yup\ \Yup\ \Yup\ \underline{a}\ \Yup\ \Yup\ \Yup\ \Yup\,\overline{a}\ \Yup\ \Yup\ \Yup\ \cdots$$

$$\Yleft\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \Yup\ \cdots$$

Exercise 2

Consider the following program and explain which function does it compute.

$$(s, \Yleft, s_1, \Yleft, R)$$
$$(s_1, 0, s_1, \Yup, R)$$
$$(s_1, 1, s_1, \Yup, R)$$
$$(s_1, \Yup, s_2, \Yup, L)$$
$$(s_2, \Yup, s_2, \Yup, L)$$
$$(s_2, \Yleft, s_3, \Yleft, R)$$
$$(s_3, \Yup, t, 1, -)$$

Exercise 3

Specify a total Turing machine that accepts its input string if its length is prime.

Hint. Give an implementation of the Sieve of Eratosthenes. To check whether n is prime, start writing down all the numbers from 2 to n in order. Then repeat: find the smallest number in the list, declare it prime, then cross off all multiples of that number. Repeat until each number in the list has been either declared prime or crossed off as a multiple of a smaller prime.

Exercise 4

Search in the WWW for Turing machine simulators. Select one, make a brief description of how it works and give some examples. Use it to simulate the behaviour of the Turing machine discussed so far.

Languages

The set $\mathsf{L}(\mathsf{M}) = \{w \in \Sigma^* \mid \mathsf{M} \text{ accepts } w\}$ is called the set (or *language*) accepted by Turing machine M. A Turing Machine is *total* if it halts (either by accepting or rejecting) in all inputs.

A language is

- *recursively enumerable* if it is the language $\mathsf{L}(\mathsf{M})$ recognised by some Turing Machine M, i.e.
$$\begin{cases} \mathsf{M} \text{ halts} & \Leftarrow w \in \mathsf{L} \\ \mathsf{M} \text{ halts in the non-acceptance state or loops forever} & \Leftarrow w \notin \mathsf{L} \end{cases}$$
- *co-recursively enumerable* if its complement is recursively enumerable;
- *recursive* if it is the language $\mathsf{L}(\mathsf{M})$ recognised by some *total* Turing Machine M, i.e.
$$\begin{cases} \mathsf{M} \text{ halts in the acceptance state} & \Leftarrow w \in \mathsf{L} \\ \mathsf{M} \text{ halts in the rejection state} & \Leftarrow w \notin \mathsf{L} \end{cases}$$

A property ϕ (over strings) is

- *decidable* if the set of all strings exhibiting ϕ is *recursive*, i.e. if there is a total Turing machine that accepts all input strings that have property ϕ and rejects those that do not.
- *semidecidable* if the set of all strings exhibiting ϕ is *recursively enumerable*, i.e. if there is a Turing machine that accepts all input strings that have property ϕ and rejects or loops if not.

Note: Properties (decidable) vs languages (recursive)

$$\begin{aligned} \phi \text{ is decidable} &\iff \{w \mid \phi(w)\} \text{ is recursive} \\ \mathsf{S} \text{ is recursive} &\iff w \in \mathsf{S} \text{ is decidable} \end{aligned}$$

Universal Turing Machines

Turing machines are not restricted to do just a single computational task but can be *programmed* to do many different ones. Actually a Turing machine can simulate other Turing machines whose descriptions are presented as part of the input.

The key issue is to fix a reasonable encoding scheme for Turing machines over the alphabet $\{0, 1\}$, e.g.

$$0^n10^m10^k10^s10^t10^u10^v1,$$

and then construct U such that

$$L(U) = \{M\#s \mid s \in L(M)\}$$

where symbol # is just a new symbol to separate M from its input. If the encodings of M and s are valid, U makes a step-by-step simulation of machine M.

This encoding scheme should be such that all the data associated with a machine M — the set of states, the transition function, the input and tape alphabets, the endmarker, the blank symbol, and the start, accept, and reject states-can be determined easily by another machine reading the encoded description of M. For example, the expression above might indicate that the machine has n states represented by the numbers 0 to $n - 1$; it has m tape symbols represented by the numbers 0 to $m - 1$, of which the first k represent input symbols; the start, accept, and reject states are s, t and T, respectively; and the end marker and blank symbol are u and v, respectively. The remainder of the string can consist of a sequence of substrings specifying the transitions. For example, the substring

$$0^a10^p10^q10^b10$$

might indicate that δ contains the transition $((p, a), (q, b, L))$, where the direction to move the head encoded by the final digit.

We may then discuss the expressive power and limitations of Turing machines using no other instruments other than the machines themselves. It is precisely this self-referential property that Gödel exploited to embed statements about arithmetics in statements of arithmetics in his Incompleteness Theorem. The embedding in the Theorem is the same as the encoding of Turing machines into input forms acceptable for universal machines and is achieved by converting the finite description of a Turing machine into a unique non-negative integer. The conversion is possible as we are only dealing here with machines having a finite number of states, a finite number of symbols in its alphabet, and only a finite number of movements for their heads.

Revisiting undecidable problems

Our quest for a precise notion of an *algorithm* is concluded with the identification

$$\boxed{\text{Algorithm} \equiv \text{a Turing machine that halts on all inputs}}$$

As discussed before, this entails the existence of some computational problems which cannot be solved by any algorithm.

Turing machines (like other formalisms such as finite or push-down automata, regular, context-free or unrestricted grammars) can be represented by sequences of symbols (cf, the discussion on universal Turing machines). As there is only a countable number of languages defined over an alphabet, the number of languages specified by Turing machines (cf, recursive and recursive enumerable languages) is also countable. Thus Turing machines decide or semi-decide on an infinitesimal fraction of all possible computational problems. According to the Church-Turing thesis, we have reached a fundamental limitation: *computational tasks that cannot be performed by a Turing machine are undecidable.*

We can now give a mathematically rigorous version of this fact (somehow paradigmatic within last century's scientific culture) by framing it in the formalism of Turing machines. Actually, we have now a formalised notion of an algorithm and a sort of universal programming language – the Turing machine. In this setting we can define a language which is not recursive e prove that indeed such is the case. Consider,

$$Y = \{"Mw" \mid \text{Turing machine } M \text{ halts on input } w\}$$

Clearly, Y is recursively enumerable: it is the language recognised by an universal Turing machine. Such a machine halts exactly when its input is in Y.

Suppose that Y is decidable by some machine N. Then, given a particular Turing machine M semideciding language $L(M)$, one could design another machine that actually *decides* $L(M)$ by writing $"M""w"$ in its input tape and then simulating N on this input. If such is the case, every recursively enumerable language would be recursive.

However, Y is not recursive. Actually, if it were recursive, language

$$Z = \{"M" \mid \text{Turing machine } M \text{ halts on input } "M"\}$$

would also be recursive (cf. put $"M""M"$ on the tape of a new machine and hand control to N). As it can be proved that recursive languages are closed for complements, it suffices to show that the complement of Z is not recursive.

$$\overline{Z} = \{w \mid \Psi(w)\}$$

where $\Psi(w) \;=\; w$ is not encoding of a Turing machine, or it is the encoding of a Turing machine M that does not halt on $"M"$.

$\overline{Z}$, finally, is not recursively enumerable, let alone recursive, Suppose that there exists a machine K semideciding $\overline{Z}$. Is $"K"$ in $\overline{Z}$?

- $"K" \in \overline{Z}$ iff K does not accept input $"K"$;
- but, K is supposed to semidecide $\overline{Z}$; so $"K" \in \overline{Z}$ iff K accepts input $"K"$.

Thus, we get an example of a non recursive language and proved an important theorem: the set of recursive languages is a strict subset of the set of recursive enumerable functions.

The halting problem can be also stated in this formalism as follows: *Is there an algorithm that decides, for an arbitrary Turing machine* M *and input* w*, whether or not* M *accepts* w*?*.

Other undecidable problems for Turing machines include

- does M halt on the empty tape?
- is there any string at all on which M halts?

- does M halts on every input?
- given two Turing machines, do they halt on the same input?
- does M fails to halt on input w?

Exercise 5

Is it decidable whether a given Turing machine

- has at least 30 states?
- accepts the null string ϵ?

Hint. To show that a problem is decidable amounts to give a total Turing machine that accepts exactly the positive instances and rejects the others. On the other hand, undecidable problems are identified by showing that a decision procedure for it could be used to construct a decision procedure for the halting problem, which does not exist.

Notes.
Both the textbook of H. Lewis and D. Papadimitriou [4] or the lecture notes by D. Kozen [3] provide excellent introductions to computability and the (classical) theory of computation. A quite interesting book by N. Yanofsky [5] may help to build up the correct intuitions which often is as important as mastering the technicalities. As a side reading on Gödel's incompleteness theorem and connections to computability I suggest reference [1]. A. Hodges biography of Alan Turing [2] makes a most pleasant weekend reading (see also the book website at `www.turing.org.uk`).

References

[1] M. Baaz, C. H. Papadimitriou, H. W. Putman, D. S. Scott, and C. L. Harper. *Kurt Godel and the Foundations of Mathematics*. Cambridge University Press, 2011.

[2] A. Hodges. *Alan Turing, the Enigma*. Princeton University Press, 1983.

[3] D. C. Kozen. *Automata and Computability*. Springer, 1999.

[4] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall (2nd Edition), 1997.

[5] N. S. Yanofsky. *The Outer Limits of Reason*. MIT Press, 2013.