Lecture 2:
Computability

MIEFis - Quantum Computation, L. S. Barbosa, 2019-20

Summary
(1) The quest for a formal definition of algorithm.
(2) Turing machines. Universal Turing machines.
(3) The Church-Turing thesis. The uncomputable; the undecidable.
(4) Recursive functions.

---

1 The question

- What computational tasks are possible?
- What does it mean for a function to be computable?
- Are there any noncomputable functions?
- How does computational power depend on programming constructs?

Models of computations vs Grammars [1]

Models	Grammars (Chomsky hierarchy)
finite memory: finite automata	right-linear grammars
finite memory with stack: pushdown automata	context-free grammars
linear bounded automata	context-sensitive grammars
unrestricted memory:	
Turing machines (Alan Turing)	
Post systems (Emil Post)	
μ-recursive functions (K. Gödel, J. Herbrand)	unrestricted grammars
λ-calculus (A. Church, S. Kleene)	
Combinatory logic (M. Schönfinkel, Haskell Curry)	

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

[1] For those wondering on the use of grammars to specify computational models, notice that symbol manipulation or parsing a sentence in a language bears a strong resemblance to computation.

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.

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 construct 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 the of formal systems, culminating on Turing's theorem that there is no algorithm to solve the halting problem.

Church-Turing thesis

All the formalisms above capture precisely the same intuition about what it means to be *effectively computable*. Or, in from more formal perspective,

> *The class of functions computable by a Turing machine corresponds exactly to the class of functions which can be regarded as being computable by an algorithm.*

2 Turing machines

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 $\prec, \succ \in \Gamma$, 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 $\succ$. 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, where H the set of halting states ($t, s \in H$):

$\forall_{p \in Q-H}.\ \delta(p, \prec) = (q, \prec, R)$	never moves off to the left of $\prec$
$\forall_{p \in Q-H}, \forall_{a \in \Gamma}.\ \delta(p, a) = (q, b, -) \Rightarrow b \neq \prec$	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 (loop on) without ever accepting or rejecting.

A configuration is a tuple in $Q \times \{w \lrcorner^{\omega} \mid w \in \Sigma^*\} \times \mathbb{N}$ 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 ($n \geq 0$). Ex. the initial configuration on input $x \in \Sigma^*$: $(s, \prec x \lrcorner^{\omega}, 0)$.

The *transition relation*:

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

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 $\perp$, 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.

$$\prec\ a\ a\ b\ b\ a\ a\ a\ b\ b\ a\ \perp\ \perp\ \perp\ \cdots$$
$$\prec\ a\ a\ b\ b\ a\ a\ a\ b\ b\ \overline{a}\ \perp\ \perp\ \perp\ \cdots$$
$$\prec\ \underline{a}\ a\ b\ b\ a\ a\ a\ b\ b\ \overline{a}\ \perp\ \perp\ \perp\ \cdots$$
$$\prec\ \underline{a}\ a\ b\ b\ a\ a\ a\ b\ \overline{b}\ \overline{a}\ \perp\ \perp\ \perp\ \cdots$$
$$\prec\ \underline{a}\ \underline{a}\ b\ b\ a\ a\ a\ b\ \overline{b}\ \overline{a}\ \perp\ \perp\ \perp\ \cdots$$
$$\prec\ \underline{a}\ \underline{a}\ b\ b\ a\ a\ a\ \overline{b}\ \overline{b}\ \overline{a}\ \perp\ \perp\ \perp\ \cdots$$
$$\cdots$$
$$\prec\ \underline{a}\ \underline{a}\ \underline{b}\ \underline{b}\ \underline{a}\ \overline{a}\ \overline{a}\ \overline{b}\ \overline{b}\ \overline{a}\ \perp\ \perp\ \perp\ \cdots$$

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.

$$
\begin{aligned}
&\prec \underline{a}\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\,\overline{a}\,\overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \lambda\ \lambda\ \lambda\ \cdots\\
&\prec \lambda\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\ \lambda\ \overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \lambda\ \lambda\ \lambda\ \cdots\\
&\prec \lambda\ \lambda\ \underline{b}\,\underline{b}\,\underline{a}\ \lambda\ \lambda\overline{b}\,\overline{b}\,\overline{a}\ \lambda\ \lambda\ \lambda\ \cdots\\
&\cdots\\
&\prec \lambda\ \lambda\ \lambda\ \lambda\ \underline{a}\ \lambda\ \lambda\ \lambda\ \lambda\overline{a}\ \lambda\ \lambda\ \lambda\ \cdots\\
&\prec \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \lambda\ \cdots
\end{aligned}
$$

Exercise 1

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

$$
\begin{aligned}
&(s, \prec, s_1, \prec, R)\\
&(s_1, 0, s_1, \lambda, R)\\
&(s_1, 1, s_1, \lambda, R)\\
&(s_1, \lambda, s_2, \lambda, L)\\
&(s_2, \lambda, s_2, \lambda, L)\\
&(s_2, \prec, s_3, \prec, R)\\
&(s_3, \lambda, t, 1, -)
\end{aligned}
$$

Exercise 2

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.

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

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.

Exercise 3

Given a decidable property, what is the corresponding recursive set? Conversely, which decidable property corresponds to a given recursive set?

Note: Properties (decidable) vs languages (recursive)

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

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.

$$0^n 1 0^m 1 0^k 1 0^s 1 0^t 1 0^u 1 0^v 1,$$

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.

3 Undecidability

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

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

This, however, opens the possibility of formally showing that there are 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. Turing machines decide or semidecide on an infinitesimal fraction of all possible languages. According to the Church-Turing thesis, we have reached a

fundamental limitation: *computational tasks that cannot be performed by a Turing machine are undecidable.*

The halting problem

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

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

```
1 if dec(P, P) then
  |   go to 1;
  else
  |   halt;
  end
```

Exercise 4

Explain why $\mathtt{dec}(\mathtt{p}, \mathtt{x})$ cannot exist and conclude that no algorithm to decide whether an arbitrary program would halt or loop does not exist. **Hint.** What is the result of $\mathtt{DecideHalt}(\mathtt{DecideHalt})$? It halts iff $\mathtt{dec}(\mathtt{DecideHalt}, \mathtt{DecideHalt})$ returns false ...

We can now give a mathematically rigorous version of this paradox (somehow paradigmatic within last century's scientific culture). Note that 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.

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

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

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

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 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: *there is no algorithm that decides, for an arbitrary Turing machine* M *and input* w*, whether or not* M *accepts* w.

Other undecidable problems for Turing machines:

- 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?

Other problems can be reduced to one of these (e.g. the tiling problem).

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.

- Build a Turing machine that, given the encoding of M written on its input tape, counts the number of states of M and accepts or rejects depending on whether the number is at least 30.

- Suppose it is possible to decide whether a given machine accepts ϵ. Thus the halting problem could be decided as follows: Suppose we are given a Turing machine M and string x to determine whether M halts on x. Construct a new machine N that does the following on input y: i) erases its input y; ii) writes x on its tape (x is hard-wired in N finite control); iii) runs M on input x (M is hard-wired in N finite control); (iv) accepts if M halts on x. Note if M halts on x, then N accepts its input y; and if M does not halt on x, then N does not halt on y, therefore does not accept y. Moreover, this is true for every y. Thus,

$$L(N) = \begin{cases} \Sigma^* & \text{if } M \text{ halts on } x \\ \emptyset & \text{if } M \text{ does not halt on } x \end{cases}$$

  Now if we know whether a given machine accepts ϵ, we could apply this decision procedure to the N, and this would tell whether M halts on x, thus obtaining a decision procedure for halting.

Exercise 6

Try some examples in a Turing machine simulator on the web. Look for such simulators written in HASKELL

---

4 Recursive functions

We turn now to an alternative approach to computability, focussed on *what is computed* rather than on an explicit *model of computation*.

Primitive recursive functions

Are all basic functions listed below

- k-ary *zero* functions: $z(n_1, \cdots, n_k) = 0$
- k-ary, j-*projection* functions: $\mathrm{id}_{k,j}(n_1, \cdots, n_k) = n_j$
- the *successor* function: $s(n) = n + 1$

and those obtained by them by any number of successive application of composition and recursive definition, i.e. by

- function *composition* of a k-ary function g with k m-ary functions h_i:

$$f(n_1, \cdots, n_m) = g(h_1(n_1, \cdots, n_m), h_2(n_1, \cdots, n_m), \cdots, h_k(n_1, \cdots, n_m))$$

- recursive definition by a k-ary g and a $k+2$-ary h function:

$$\begin{aligned} f(n_1, \cdots, n_k, 0) &= g(n_1, \cdots, n_k) \\ f(n_1, \cdots, n_k, m+1) &= h(n_1, \cdots, n_k, m, f(n_1, \cdots, n_k, m)) \end{aligned}$$

Exercise 7

Show that

$$\mathrm{add}(m, n) \;=\; m + n$$

is primitive recursive.

Hint. The function is recursively defined from functions obtained by combining the identity, zero and successor functions. Make above $k = 1$, $g = \mathrm{id}_{1,1}$ and $h(m, n, p) = \mathrm{succ}(\mathrm{id}_{3,3}(m, n, p)$, yielding

$$\begin{aligned} \mathrm{add}(m, 0) &= m \\ \mathrm{add}(m, n+1) &= \mathrm{succ}(\mathrm{add}(m, n)) \end{aligned}$$

Exercise 8

Show that if $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ is primitive recursive, so is function

$$sum_f(n, m) = \sum_{i \in \{0,1,\cdots,m\}} f(n, i)$$

Exercise 9

Show that the set of primitive recursive functions is countable. Use this fact to prove that most functions in $\mathbb{N}$ are not primitive recursive, and therefore, according to Church thesis, uncomputable.

Hint. List all unary primitive recursive functions, as strings, in lexicographic order:

$$f_1, f_2, f_3, \cdots$$

In principle, given n we can find f_n in the list and use it to compute the natural number $z(n) = f_n(n)+1$. Clearly, $z(n)$ is computable (we just did it!), but still it is not a primitive recursive function. Why?

If it were a primitive recursive function, for exemple $z = f_m$ for some m, then

$$f_m(m) + 1 = f_m(m)$$

which is absurd.

Exercise 10

Show that the factorial function, the function that computes the greatest common divisor and the prime predicate (which returns 1 if its argument is a prime number) are all primitive recursive functions.

μ-recursive functions

Let p be a $k+1$-ary function and define the following *unbounded iteration* scheme:

$$\mu_m[p(n_1, \cdots, n_k, m) = 1] \;=\; \begin{cases} \text{the least } m \text{ such that } p(n_1, \cdots, n_k, m) = 1 \;\Leftarrow \text{ such a } m \text{ exists} \\ 0 \;\Leftarrow \text{ otherwise} \end{cases}$$

The obvious way to compute this function is through unbounded iteration, i.e. through a `while` loop:

$m := 0$;
while $p(n_1, \cdots, n_k, m) \neq 1$ **do**
| $m := m + 1$
end
return m

This, however, may fail to terminate. We call function p *minimalizable* if the minimization scheme above terminates for every input.

Functions defined by the basic functions above, composition, recursion and minimization are called μ-*recursive*.

Example

Consider function $\log(b, n)$ standing for the logarithm of $n+1$ over base $b+2$. Formally,

$$\log(b, n) \;=\; \mu_m[geq(b+2)^m, n+1)]$$

Exercise 11

Why are we using $b+2$ and $n+1$ above? Show that the minimization algorithm terminates, i.e. function

$$f(b, n, m) \;=\; geq((b+2)^m, n+1)$$

is minimizable.

Theorem

A function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ is μ-recursive iff it is recursive, i.e. computable by a Turing machine.

Exercise 12

Suppose that g is a μ-recursive bijection over $\mathbb{N}$. Show that its inverse g^{-1} is μ-recursive as well.

Notes.
Both the textbook of H. Lewis and D. Papadimitriou [6] or the lecture notes by D. Kozen [5] provide excellent introductions to computability and the (classical) theory of computation. A quite interesting book by N. Yanofsky [8] 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 [2]. A. Hodges biography of Alan Turing [4] makes a most pleasant weekend reading (see also the book website at `www.turing.org.uk`).

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] 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.

[3] D. Z. Du and K. I. Ko. *Theory of Computational Complexity*. Addison-Wesley, 2000.

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

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

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

[7] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

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