Computability and Computational Complexity
Academic year 2019–2020, first semester
Lecture notes

Mauro Brunato

Version: 2019-11-08

Caveat Lector

The main purpose of these very schematic lecture notes is to keep track of what has been said during the lectures. Reading this document is not enough to pass the exam. You should also see the linked citations and footnotes, the additional material and the references provided on the following webpage, which will also contain the up-to-date version of these notes:

`https://comp3.eu/`

To check for new version of this document, please compare the version date on the title page the one reported on the webpage.

Changelog

2019-11-08

- Characterization of class NP; examples of NP problems: SATISFIABILITY, CLIQUE, TSP.
- Polynomial reductions, examples for known NP problems.
- NP-hard and NP-complete definitions.
- Boolean circuits and CNF representations of a circuit.
- Exercise on Boolean circuits.

2019-11-05

- Computational complexity: definitions.
- Classes DTIME(f), **P**, **EXP**.
- Non-deterministic Turing machines, class **NP** (to be continued).

2019-10-28

- Post Correspondence Problem (completed).
- Kolmogorov complexity (small introduction).

2019-10-24

- Rice's theorem and exercises.
- Post Correspondence Problem (to be continued).

2019-10-12

- Universal Turing machines, uncomputability results.
- Busy beaver functions and results.
- Turing reductions
- Self-assessment questions and exercises.

2019-10-01

- First lectures: basic definitions, Collatz example, Turing machines, computational power of Turing machines.

Contents

Part I

Lecture notes

Chapter 1

Computability

1.1 Basic definitions and examples

In computer science, every problem instance can be represented by a finite sequence of symbols from a finite alphabet, or equivalently as a natural number. In the following, let Σ denote a finite set of *symbols.* Σ will be the *alphabet* we are going to use to represent things. Pairs, triplets, n-tuples of symbols are represented by the usual cartesian product notations:

$$\Sigma^2 = \Sigma \times \Sigma = \{(s,t)|s,t \in \Sigma\}, \quad \Sigma^3 = \Sigma \times \Sigma \times \Sigma, \dots, \Sigma^n = \overbrace{\Sigma \times \Sigma \times \cdots \times \Sigma}^{n \text{ times}}$$

As a shorthand, instead of representing tuples of symbols in the formal notation $(s_1, s_2, \dots, s_n)$ we will use the simpler "string" notation $s_1 s_2 \cdots s_n$. As a particular case, let $\varepsilon = ()$ represent the empty tuple (with $n = 0$ elements). Therefore, the set of strings of length n can be defined by induction:

$$\Sigma^n = \begin{cases} \{\varepsilon\} & \text{if } n = 0 \\ \Sigma \times \Sigma^{n-1} & \text{if } n > 0. \end{cases}$$

Finally, the Kleene closure of this sequence is the set of all *finite* strings on the alphabet Σ:

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n.$$

It is worthwile to note that Σ^*, while being infinite in itself, only contains *finite* sequences of symbols. Moreover, for every $n \in \mathbb{N}$, Σ^n is finite ($|\Sigma^n| = |\Sigma|^n$, where $|\cdot\|$ represents the cardinality of a set).

Our main focus will be on functions that map input strings to output strings on a given alphabet,

$$f : \Sigma^* \to \Sigma^*,$$

or in functions that map strings onto a "yes"/"no" decision set,

$$f : \Sigma^* \to \{0, 1\};$$

in such case, we talk about a *decision problem.*

Examples:

- Given a natural number n, is n prime?
- Given a graph, what is the maximum degree of its nodes?
- From a customer database, select the customers that are more than fifty years old.

- Given a set of pieces of furniture and a set of trucks, can we accommodate all the furniture in the trucks?

As long as the function's domain and codomain are finite, they can be represented as sequences of symbols, hence of bits, therefore as strings (although some representations make more sense than others); observe that some problems among those listed are decision problems, others aren't.

Decision functions and sets

There is a one-to-one correspondence between decision functions on Σ^* and subsets of Σ^*. Given $f : \Sigma^* \to \{0, 1\}$, its obvious set counterpart is the subset of strings for which the function answers 1:

$$S_f = \{s \in \Sigma^* : f(s) = 1\}.$$

Conversely, given a string subset $S \subseteq \Sigma^*$, we can always define the function that decides over elements of the set:

$$f_S(s) = \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{if } s \notin S. \end{cases}$$

Given a function, or equivalently a set, we say that it is **computable**[1] (or **decidable**, or **recursive**) if and only if a procedure can be described to compute the function's outcome in a finite number of steps. Observe that, in order for this definition to make sense, we need to define what an acceptable "procedure" is; for the time being, let us intuitively consider any computer algorithm.

Examples of computable functions and sets are the following:

- the set of even numbers;
- a function that decides whether a number is prime or not;
- any finite or cofinite[2] set, and any function that decides on them;
- any function studied in a basic Algorithms course (sorting, hashing, spanning trees on graphs. . .).

1.1.1 A possibly non-recursive set

Collatz numbers

Given $n \in \mathbb{N} \setminus \{0\}$, let the *Collatz sequence* starting from n be defined as follows:

$$\begin{aligned} a_1 &= n \\ a_{i+1} &= \begin{cases} a_i/2 & \text{if } a_i \text{ is even} \\ 3a_i + 1 & \text{if } a_i \text{ is odd,} \end{cases} \qquad i = 1, 2, \dots \end{aligned}$$

In other words, starting from n, we repeatedly halve it while it is even, and multiply it by 3 and add 1 if it is odd.

The *Collatz conjecture*[3] states that every Collatz sequence eventually reaches the value 1. While most mathematicians believe it to be true, nobody has been able to prove it.

Suppose that we are asked the following question:

"Given $n \in \mathbb{N} \setminus \{0\}$, does the Collatz sequence starting from n reach 1?"

[1] `https://en.wikipedia.org/wiki/Recursive_set`
[2] A set is *cofinite* when its complement is finite.
[3] `https://en.wikipedia.org/wiki/Collatz_conjecture`

```
function collatz ($n \in \mathbb{N} \setminus \{0\}$) $\in \{0,1\}$
    repeat
        if $n = 1$ then return 1
        if $n$ is even
            then $n \leftarrow n/2$
            else $n \leftarrow 3n+1$
    return 0
```

```
function collatz ($n \in \mathbb{N} \setminus \{0\}$) $\in \{0,1\}$
    return 1
```

Figure 1.1: Left: the only way I know to decide whether n is a Collatz number isn't guaranteed to work. Right: a much better way, but it is correct if and only if the conjecture is true.

If the answer is "yes," let us call n a *Collatz number*. Let $f : \mathbb{N} \setminus \{0\} \to \{0,1\}$ be the corresponding decision function:

$$f(n) = \begin{cases} 1 & \text{if } n \text{ is a Collatz number} \\ 0 & \text{if } n \text{ is not a Collatz number,} \end{cases} \qquad n = 1, 2, \dots$$

Then the Collatz conjecture simply states that all positive integers are Collatz numbers or, equivalently, that $f(n) = 1$ on its whole domain.

Decidability of the Collatz property

Let us consider writing a function, in any programming language, to answer the above question, i.e., a function that returns 1 if and only if its argument is a Collatz number. Figure1.1 details two possible ways to do it, and both have problems: the rightmost one requires us to have faith in an unproven mathematical conjecture; the left one only halts when the answer is 1 (the final **return** is never reached).

In more formal terms, we are admitting that we are **not** able to prove that the Collatz property is *decidable* (i.e., that there is a computer program that always terminates with the correct answer[4]). However, we have provided a procedure that terminates with the correct answer when the answer is "yes" (the function is not *total*, in the sense that it doesn't always provide an answer). We call such set **recursively enumerable**[5] (or RE, in short).

Having a procedure that only terminates when the answer is "yes" maight not seem much, but it actually allows us to enumerate all numbers having the property. The function in Fig. 1.2 shows the basic trick to enumerate a potentially non-recursive set, applied to the Collatz sequence: the **diagonal method**[6]. Rather than performing the whole decision function on a number at a time (which would expose us to the risk of an endless loop), we start by executing the first step of the decision function for the first input ($n = 1$), then we perform the second step for $n = 1$ and the first step of $n = 2$; at the i-th iteration, we perform the i-th step of the first input, the $(i-1)$-th for the second, down to the first step for the i-th input. This way, every Collatz number will eventually hit 1 and be printed out.

The naïf approach of following the table rows is not guaranteed to work, since it would loop indefinitely, should a non-Collatz number ever exist.

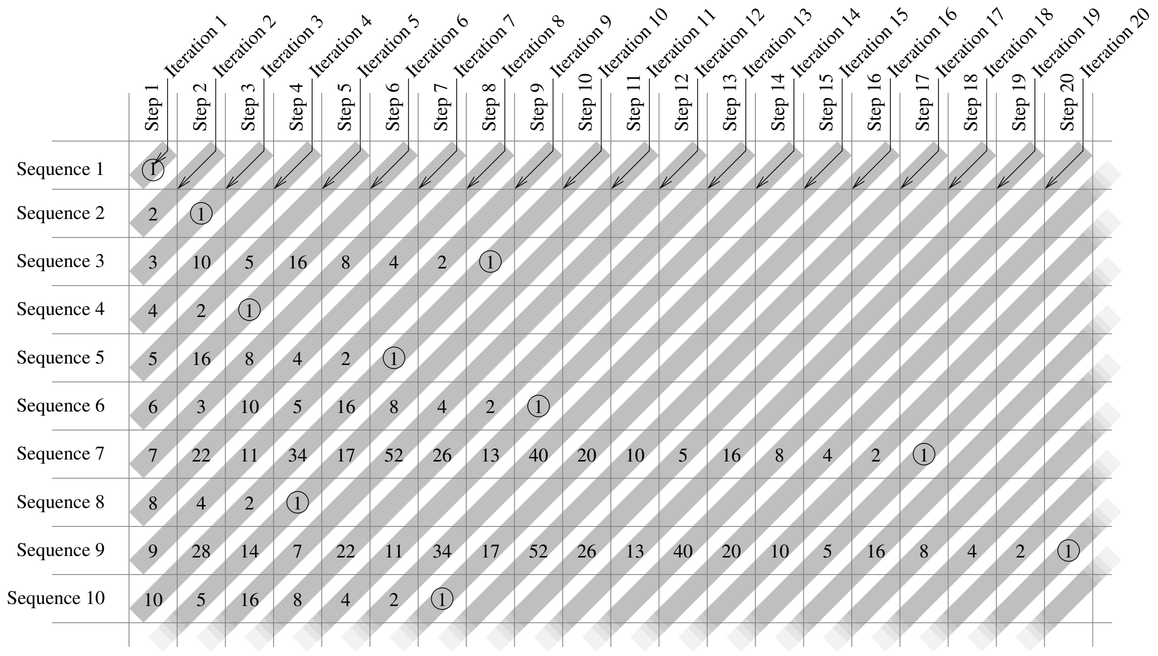
Observe that the procedure does not print out the numbers in increasing order.

1.2 A computational model: the Turing machine

Among the many formal definition of computation proposed since the 1930s, the Turing Machine (TM for short) is by far the most similar to our intuitive notion. A Turing Machine[7] is defined by:

[4]To the best of my knowledge, which isn't much.
[5]`https://en.wikipedia.org/wiki/Recursively_enumerable_set`
[6]See `https://comp3.eu/collatz.py` for a Python version.
[7]`https://en.wikipedia.org/wiki/Turing_machine`

```
procedure enumerate_collatz
    queue ← []
    for $n$ ← 1 ... ∞                                   Repeat for all numbers
        queue$_n$ ← $n$                                  Add n to queue with itself as starting value
        for $i$ ← 1 ... $n$:                             Iterate on all numbers up to n
            if queue$_i$ = 1                             i is Collatz, print and forget it
                print $i$
                delete queue$_i$                         deleted means "Already taken care of"
            else if queue$_i$ is not deleted             if current number wasn't printed and forgotten yet
                if queue$_i$ is even                     Advance i-th sequence in the queue by one step
                    then queue$_i$ ← queue$_i$ / 2
                    else queue$_i$ ← 3 · queue$_i$ + 1
```



	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11	Step 12	Step 13	Step 14	Step 15	Step 16	Step 17	Step 18	Step 19	Step 20
Sequence 1	①																			
Sequence 2	2	①																		
Sequence 3	3	10	5	16	8	4	2	①												
Sequence 4	4	2	①																	
Sequence 5	5	16	8	4	2	①														
Sequence 6	6	3	10	5	16	8	4	2	①											
Sequence 7	7	22	11	34	17	52	26	13	40	20	10	5	16	8	4	2	①			
Sequence 8	8	4	2	①																
Sequence 9	9	28	14	7	22	11	34	17	52	26	13	40	20	10	5	16	8	4	2	①
Sequence 10	10	5	16	8	4	2	①													

Figure 1.2: Enumerating all Collatz numbers: top: the algorithm; bottom: a working schematic

- a finite alphabet Σ, with a distinguished "default" symbol (e.g., "␣" or "0") whose symbols are to be read and written on an infinitely extended tape divided into cells;
- a finite set of states Q, with a distinguished initial state and one or more distinguished halting states;
- a set of rules R, described by a (possibly partial) function that associates to a pair of symbol and state a new pair of symbol and state plus a direction:

$$R : Q \times \Sigma \to \Sigma \times Q \times \{L, R\}.$$

Initially, all cells contain the default symbol, with the exception of a finite number; the non-blank portion of the tape represent the *input* of the TM. The machine also maintains a *current position* on the tape. The machine has an initial state $q_0 \in Q$. At every step, if the machine is in state $q \in Q$, and the symbol $\sigma \in \Sigma$ appears in the current position of the tape, the machine applies the rule set R to (q, σ):

$$(\sigma', q', d) = R(q, \sigma).$$

The machine writes the symbol σ' on the current tape cell, enters state q', and moves the current position by one cell in direction d. If the machine enters one of the distinguished halting states, then the computation ends. At this point, the content of the (non-blank portion of) the tape represents the computation's *output*.

Observe that the input size for a TM is unambiguously defined: the size of the portion of tape that contains non-default symbols. Also the "execution time" is well understood: it is the number of steps before halting. Therefore, when we say that the computational complexity of a TM for inputs of size n is $T(n)$ then we mean that $T(n)$ is the worst-case number of steps that a TM performs before halting when the input has size n.

1.2.1 Examples

In order to experiment with Turing machines, many web-based simulators are available. The two top search results for "turing machine demo" are

- `http://morphett.info/turing/turing.html`
- `https://turingmachinesimulator.com/`.

Students are invited to read the simplest examples and to try implementing a TM for some simple problem (e.g., some arithmetic or logical operation on binary or unary numbers). Also, see the examples provided in the course web page.

1.2.2 Computational power of the Turing Machine

With reference to more standard computational models, such as the Von Neumann architecture of all modern computers, the TM seems very limited; for instance, it lacks any random-access capability. The next part of this course is precisely meant to convince ourselves that a TM is exactly as powerful as any other (theoretical) computational device. To this aim, let us discuss some possible generalizations.

Multiple-tape Turing machines

A k-tape Turing machine is a straightforward generalization of the basic model, with the following variations:

- the machine has k unlimited tapes, each with an independent current position;

- the rule set of the machine takes into account k symbols (one for each tape, from the current position) both in reading and in writing, and k movement directions (each current position is independent), with the additional provision of a "stay" direction S in which the position does not move:
$$R : Q \times \Sigma^k \to \Sigma^k \times Q \times \{L, R, S\}^k.$$

Multiple-tape TMs are obviously more practical for many problems. For example, try following the execution of the binary addition algorithms below:

- 1-tape addition from `http://morphett.info/turing/turing.html`: select "Load an example program/Binary addition";
- 3-tape addition from `https://turingmachinesimulator.com/`: select "Examples/3 tapes/Binary addition".

However, it turns out that any k-tape Turing machine can be "simulated" by a 1-tape TM, in the sense that it is possible to represent a k-tape TM on one tape, and to create a set of 1-tape rules that simulates the evolution of the k-tape TM. Of course, the 1-tape machine is much slower, as it needs to repeatedly scan its tape back and forth just to simulate a single step of the k-tape one.

Theorem 1 (k-tape Turing machine emulation)**.** *If a k-tape Turing machine $\mathcal{M}$ takes time $T(n)$ on inputs of time n, then it is possible to program a 1-tape Turing machine $\mathcal{M}'$ that simulates it (i.e., essentially performs the same computation) in time $O(T(n)^2)$.*

Proof. See Arora-Barak, Claim 1.9 in the public draft.

Basically, the k tapes of $\mathcal{M}$ are encoded on the single tape of $\mathcal{M}'$ by alternating the cell contents of each tape; in order to remember the "current position" on each tape, every symbol is complemented by a different version (e.g., a "hatted" symbol) to be used to mark the current position. To emulate a step of $\mathcal{M}$, the whole tape of $\mathcal{M}'$ is first scanned in order to find the k symbols in the current positions; then, a second scan is used to replace each symbol in the current position with the new symbol; then a third scan performs an update of the current positions.

Since $\mathcal{M}$ halts in $T(n)$ steps, no more that $T(n)$ cells of the tapes will ever be visited; therefore, every scan performed by $\mathcal{M}'$ will take at most $kT(n)$ steps. Given some more details, cleanup tasks and so on, the simulation of a single step of $\mathcal{M}$ will take at most $5kT(n)$ steps by $\mathcal{M}'$, therefore the whole simulation takes $5kT(n)^2$ steps. Since $5k$ is constant wrt the input size n, the result follows. □

Size of the alphabet

The number of symbols that can be written on a tape (the size of the alphabet Σ) can make some tasks easier; for instance, in order to deal with binary numbers a three-symbol alphabet ("0", "1", and the blank as a separator) is convenient, while working on words is easier if the whole alphabet is available.

While a 1-sized alphabet $\Sigma = \{\text{␣}\}$ is clearly unfit for a TM (no way to store information on the tape), a 2-symbol alphabel is enough to simulate any TM:

Theorem 2 (Emulation by a two-symbol Turing Machine)**.** *If a Turing machine $\mathcal{M}$ with a k-symbol alphabet Σ takes time $T(n)$ on an input of size n, then it can be simulated by a Turing machine $\mathcal{M}'$ with a 2-symbol alphabet $\Sigma' = \{0, 1\}$ in time $O(T(n))$ (i.e., with a linear slowdown).*

Proof. See Arora-Barak, claim 1.8 in the public draft, where for convenience machine $\mathcal{M}'$ is assumed to have 4 symbols and the tape(s) extend only in one direction.

Every symbol from alphabet Σ can be encoded by $\lceil \log_2 k \rceil$ binary digits from Σ'. Every step of machine $\mathcal{M}$ will be simulated by $\mathcal{M}'$ by reading $\lceil \log_2 k \rceil$ cells in order to reconstruct the current symbol in $\mathcal{M}$; the symbol being reconstructed bit by bit is stored in the machine state (therefore, $\mathcal{M}'$ requires many more states that $\mathcal{M}$). This scan is followed by a new scan to replace the encoding with the new symbol (again, all information needed by $\mathcal{M}'$ will be "stored" in its state), and a third (possibly longer) scan to place the current position to the left or right encoding. Therefore, a step of $\mathcal{M}$ will require less than $4\lceil\log_2 k\rceil$ steps of $\mathcal{M}'$, and the total simulation time will be

$$T'(n) \leq 4\lceil\log_2 k\rceil T(n).$$

□

Simulating other computational devices

Although they are very simple devices, we can convince ourselves quite easily that Turing machines can emulate a simple CPU/RAM architecture: just replace random access memory with sequential search on a tape (tremendous slowdown, but we are not concerned by it now), the CPU's internal registers can be stored in separate tapes, and every opcode of the CPU corresponds to a separate set of states of the machine. Operations such as "load memory to a register," "perform an arithmetic or logical operation between registers," "conditionally junp to memory" and so on can be emulated.

1.2.3 Universal Turing machines

The main drawback of TMs, as described up to now, with respect to our modern understanding of computational systems, is that each serves one specific purpose, encoded in its rule set: a machine to add numbers, one to multiply, and so on.

However, it is easy to see that a TM can be represented by a finite string in a finite alphabet: each transition rule can be seen as a quintuplet, each from a finite set, and the set of rules is finite. Therefore, it is possible to envision a TM $\mathcal{U}$ that takes another TM $\mathcal{M}$ as input on its tape, properly encoded, together with an input string s for $\mathcal{M}$, and simulates $\mathcal{M}$ step by step on input s. Such machine is called a Universal Turing machine (UTM).

One such machine, using a 16 symbol encoding and a single tape, is described in

`https://www.dropbox.com/sh/u7jsxm232giwown/AADTRNqjKBIe_QZGyicoZWjYa/utm.pdf`

and can be seen in action at the aforementioned link `http://morphett.info/turing/turing.html`, clicking "Load an example program / Universal Turing machine."

1.2.4 The Church-Turing thesis

We should be convinced, by now, that TMs are powerful enough to be a fair computational model, at least on par with any other reasonable definition. We formalize this idea into a sort of "postulate," i.e., an assertion that we will assume to be true for the rest of this course.

Postulate 1 (Church-Turing thesis)**.** *Turing machines are at least as powerful as every physically realizable model of computation.*

This thesis allows us to extend every the validity negative result about TMs to every physical computational device.

1.3 Uncomputable functions

It is easy to understand that, even if we restrict our interest to decision functions, almost all functions are not computable by a TM. In fact, as the following Lemmata 1 and 2 show, there are simply too many functions to be able to define a TM for each of them.

Lemma 1. *The set of decision functions $f : \mathbb{N} \to \{0,1\}$ (or, equivalently, $f : \Sigma^* \to \{0,1\}$), is uncountable.*

Proof. By contradiction, suppose that a complete mapping exists from the naturals to the set of decision functions; i.e., there is a mapping $n \mapsto f_n$ that enumerates all functions. Define function $\hat{f}(n) = 1 - f_n(n)$. By definition, function $\hat{f}$ differs from f_n on the value it is assigned for n (if $f_n(n) = 0$ then $\hat{f}(n) = 1 - f_n(n) = 1 - 0 = 0$, and vice versa). Therefore, contrary to the assumption, the enumeation is not complete because it excluded function $\hat{f}$. □

Lemma 1 is an example of *diagonal argument*, introduced by Cantor in order to prove the uncountability of real numbers: focus on the "diagonal" values (in our case $f_n(n)$, by using the same number as function index and as argument), and make a new object that systematically differs from all that are listed.

Lemma 2. *Given a finite alphabet Σ, the number of TMs on that alphabet is countable.*

Proof. As shown in the Universal TM discussion, every TM can be encoded in some appropriate alphabet. As shown by Theorem 2, every alphabet with at least two symbols can emulate and be emulated by every other alphabet. Therefore, it is possible to define a representation of any TM in any alphabet.

We know that strings can be enumerated: first we count the only string in Σ^0, then the strings in Σ^1, then those in Σ^2 (e.g., in lexicographic order), and so on. Since every string $s \in \Sigma^*$ is finite ($s \in \Sigma^{|s|}$), sooner or later it will be enumerated. Therefore there is a mapping $\mathbb{N} \to \Sigma^*$, i.e., Σ^* is countable.

Since TMs can be mapped on a subset of Σ^* (those strings that define TMs according to the chosen encoding), and are still infinite, it follows that TMs are countable. □

Therefore, whatever way we choose to enumerate TMs and to associate them with decision functions, we will inevitably leave out some functions. Hence, given that TMs are our definition of computing,

Corollary 1. *There are uncomputable decision functions.*

1.3.1 Finding an uncomputable function

Let us introduce a little more notation. As already defined, the alphabet Σ contains a distinguished, "default" symbol, which we assume to be "␣". Before the computation starts, only a finite number of cell tapes have non-blank symbols. Let us define as "input" the smallest, contiguous set of tape cells that contains all non-blank symbols.

A Turing machine transforms an input string into an output string (the smallest contiguous set of tape cells that contain all non-blank symbols at the *end* of the computation), but it might never terminate. In other words, if we see a TM machine as a function from Σ^* to Σ^* it might not be a *total* function.

As an alternative, we may introduce a new value, ∞, as the "value" of a non-terminating computation; given a Turing machine $\mathcal{M}$, if its compuattion on input s does not terminate we will write $\mathcal{M}(s) = \infty$.

While TM encodings have a precise syntax, so that not all strings in Σ^* are syntactically valid encodings of some TM, we can just accept the convention that any such invalid string encodes the TM that immediately halts (think of s as a program, executed by a UTM that immediately stops if there is a syntax error). This way, all strings can be seen to encode a TM, and most string just encode the "identity function" (a machine that halts immediately leaves its input string unchanged). Let us therefore call $\mathcal{M}_s$ the TM whose encoding is string s, or the machine that immediately terminates if s is not a valid encoding.

With this convention in mind, we can design a function whose outcome differs from that of any TM. We employ a diagonal technique akin to the proof of Lemma 1: for any string $\alpha \in \Sigma*$, we define our function to differ from the output of the TM encoded by α on input α itself.

Theorem 3. *Given an alphabet Σ and a encoding $\alpha \mapsto \mathcal{M}_\alpha$ of TMs in that alphabet, the function*

$$UC(\alpha) = \begin{cases} 0 & \textit{if } \mathcal{M}_\alpha(\alpha) = 1 \\ 1 & \textit{otherwise} \end{cases} \qquad \forall \alpha \in \Sigma^*$$

is uncomputable.

Proof. Let $\mathcal{M}$ be any TM, and let $m \in \Sigma^*$ be its encoding (i.e., $\mathcal{M} = \mathcal{M}_m$). By definition, $UC(m)$ differs from $\mathcal{M}(m)$: the former outputs one if and only if the latter outputs anything else (or does not terminate).
See also Arora-Barak, theorem 1.16 in the public draft. □

What is the problem that prevents us from computing UC? While the definition is quite straightforward, being able to emulate the machine $\mathcal{M}_\alpha$ on input α is not enough to always decide the value of $UC(\alpha)$. We need to take into account also the fact that the emulation might never terminate. This allows us to prove, as a corollary of the preceding theorem, that there is no procedure that always determines whether a machine will terminate on a given input.

Theorem 4 (Halting problem)**.** *Given an alphabet Σ and a encoding $\alpha \mapsto \mathcal{M}_\alpha$ of TMs in that alphabet, the function*

$$HALT(s,t) = \begin{cases} 0 & \textit{if } \mathcal{M}_s(t) = \infty \\ 1 & \textit{otherwise} \end{cases} \forall (s,t) \in \Sigma^* \times \Sigma^*$$

(i.e., which returns 1 if and only if machine $\mathcal{M}_s$ halts on input t) is uncomputable.

Proof. Let's proceed by contradiction. Suppose that we have a machine $\mathcal{H}$ which computes $HALT(s,t)$ (i.e., when run on a tape containing a string s encoding a TM and a string t, always halts returning 1 if machine $\mathcal{M}_s$ would halt when run on input t, and returning 0 otherwise). Then we could use $\mathcal{H}$ to compute function UC.
For convenience, let us compute UC using a machine with two tapes. The first tape is read-only and contains the input string $\alpha \in \Sigma^*$, while the second will be used as a work (and output) tape. To compute UC, the machine will perform the following steps:

- Create two copies of the input string α onto the work tape, separated by a blank (we know we can do this because we can actually write the machine);
- Execute the machine $\mathcal{H}$ (which exists by hypothesis) on the work tape, therefore calculating whether the computation $\mathcal{M}_\alpha(\alpha)$ would terminate or not. Two outcomes are possible:
  - If the output of $\mathcal{H}$ is zero, then we know that the computation of $\mathcal{M}_\alpha(\alpha)$ wouldn't terminate, therefore, by definition of function UC, we can output 1 and terminate.
  - If, on the other hand, the output of $\mathcal{H}$ is one, then we know for sure that the computation $\mathcal{M}_\alpha(\alpha)$ would terminate, and we can emulate it with a UTM $\mathcal{U}$ (which we know to exist) and then "inverting" the result à la UC, by executing the following steps:
    - As in the first step, create two copies of the input string α onto the work tape, separated by a blank;
    - Execute the UTM $\mathcal{U}$ on the work tape, thereby emulating the computation $\mathcal{M}_\alpha(\alpha)$;
    - At the end, if the output of the emulation was 1, then replace it by a 0; if it was anything other than 1, replace it with 1.

This machine would be able to compute UC by simply applying its definition, but we know that UC is not computable by a TM; all steps, apart from $\mathcal{H}$, are already known and computable. We must conclude that $\mathcal{H}$ cannot exist.
See also Arora-Barak, theorem 1.17 in the public draft. □

This proof employs a very common technique of CS, called *reduction*: in order to prove the impossibility of $HALT$, we "reduce" the computation of UC to that of $HALT$; since we know that the former is impossible, we must conclude that the latter is too.

The Haliting Problem for machines without an input

Consider the special case of machines that do not work on an input string; i.e., the class of TMs that are executed on a completely blank tape. Asking whether a computation without input will eventually halt might seem a simpler question, because we somehow restrict the number of machines that we are considering.

Let us define the following specialized halting function:

$$HALT_\varepsilon(s) = HALT(s,\varepsilon) = \begin{cases} 0 & \text{if } \mathcal{M}_s(\varepsilon) = \infty \\ 1 & \text{otherwise} \end{cases} \forall s \in \Sigma^*.$$

It turns out that if we were able to compute $HALT_\varepsilon$ then we could also compute $HALT$:

Theorem 5. *$HALT_\varepsilon$ is not computable.*

Proof. By contradiction, suppose that there is a machine $\mathcal{H}'$ that computes $HALT_\varepsilon$. Such machine would be executed on a string s on the tape, and would return 1 if the machine encoded by s would halt when run on an empty tape, 0 otherwise.
Now, suppose that we are asked to compute $HALT(s,t)$ for a non-empty input string t. We can transform the computation $\mathcal{M}_s(t)$ on a computation $\mathcal{M}_{s'}(\varepsilon)$ on an empty tape where s' contains the whole encoding s, but prepended with a number of states that write the string t on the tape. In other words, we transform a computation on a generic input into a computation on an empty tape that writes the desired input before proceeding.
After modifying the string s into s' on tape, we can run $\mathcal{H}'$ on it. The answer of $\mathcal{H}'$ is precisely $HALT(s,t)$, which would therefore be computable. □

Again, the result was obtained by reducing a known impossible problem, $HALT$ to the newly introduced one, $HALT_\varepsilon$.

1.3.2 Recursive enumerability of halting computations

Although $HALT$ is not computable, it is clearly recursively enumerable. In fact, we can just take a UTM and modify it to erase the tape and write "1" whenever the emulated machine ends, and we would have a partial function that always accepts (i.e., returns 1) on terminating computations.

It is also possible to output all $(s,t) \in \Sigma^* \times \Sigma^*$ pairs for which $\mathcal{M}_s(t)$ halts by employing a diagonal method similar to the one used in Fig. 1.2[8].

Function $HALT$ is our first example of R.E. function that is provably not recursive.

Observe that, unlike recursivity, R.E. does *not* treat the "yes" and "no" answer in a symmetric way. We can give the following:

Definition 1. *A decision function $f : \Sigma^* \to \{0,1\}$ is* co-R.E. *if it admits a TM $\mathcal{M}$ such that $\mathcal{M}(s)$ halts with output 0 if and only if $f(s) = 0$.*

In other words, co-R.E. functions are those for which it is possible to compute a "no" answer, while the computation might not terminate if the answer is "yes". Clearly, if f is R.E., then $1 - f$ is co-R.E.

Theorem 6. *A decision function $f : \Sigma^* \to \{0,1\}$ is recursive if and only if it is both R.E. and co-R.E.*

[8]See the figure at `https://en.wikipedia.org/wiki/Recursively_enumerable_set#Examples`

Proof. Let us prove the "only if" part first. If f is recursive, then there is a TM $\mathcal{M}_f$ that computes it. But $\mathcal{M}_f$ clearly satisfies both the R.E. definition ($\mathcal{M}_f(s)$ halts with output 1 if and only if $f(s) = 1$) and the co-R.E. definition ($\mathcal{M}_f(s)$ halts with output 0 if and only if $f(s) = 0$).
About the "if" part, if f is R.E., then there is a TM M_1 such that $M_1(s)$ halts with output 1 iff $f(s) = 1$; since f is also co-R.E., then there is also a TM $\mathcal{M}_0$ such that $M_1(s)$ halts with output 0 iff $f(s) = 0$. Therefore, a machine that alternates one step of the execution of $\mathcal{M}_1$ with one step of $\mathcal{M}_0$, halting when one of the two machines halts and returning its output, will eventually terminate (because, whatever the value of f, at least one of the two machines is going to eventually halt) and precisely decides f. □

Observe that, as already pointed out, any definition given on decision functions with domain Σ^* also works on domain $\mathbb{N}$ (and on any other discrete domain), and can be naturally extended on subsets of strings or natural numbers. We can therefore define a set as recursive, recursively enumerable, or co-recursively enumerable.

Decision and acceptance

In the following, we will use the following terms when speaking of languages.

Definition 2.

- *If language S is recursively enumerable, i.e. there is a TM $\mathcal{M}$ such that $\mathcal{M}(s) = 1 \Leftrightarrow s \in S$, then we say that $\mathcal{M}$* accepts *S (or that it "recognizes" it).*
- *Given a TM $\mathcal{M}$, the language recognized by it (i.e., the set of all inputs that are accepted by the machine) is represented by $L(\mathcal{M})$.*
- *If language S is recursive, i.e. there is a TM $\mathcal{M}$ that accepts it and always halts, then we say that $\mathcal{M}$* decides *S.*

In the case of functions transforming strings, we will use the following terms.

Definition 3. *If a function $f : \Sigma^* \to \Sigma^*$ is computable, i.e. there is a TM $\mathcal{M}$ that always halts and such that $\mathcal{M}(s) = f(s)$, then we say that $\mathcal{M}$* computes *f.*

We generalize the notion to functions outside the realm of strings by considering suitable representations. E.g., a machine $\mathcal{M}$ *computes* an integer function $f : \mathbb{N} \to \mathbb{N}$ if it transforms a representation of $n \in \mathbb{N}$ (e.g., its decimal, binary or unary notation) into the corresponding representation of $f(n)$. Since all representations of integer numbers can be converted to each other by a TM, the choice of a specific one is arbitrary and does not impact on the definition. Therefore, we can resort to unary notation and say that

Theorem 7. *A function $f : \mathbb{N} \to \mathbb{N}$ is computable if and only if there is a TM $\mathcal{M}$ on alphabet $\Sigma = \{1, \text{␣}\}$ such that*

$$\forall n \in \mathbb{N} \quad \mathcal{M}(1^n) = 1^{f(n)}.$$

I.e., the TM $\mathcal{M}$ maps a string of n ones into a string of $f(n)$ ones.

1.3.3 Another uncomputable function: the Busy Beaver game

Since we might be unable to tell at all whether a specific TM will halt, the question arises of how complex can machine's output be for a given number of states.

Definition 4 (The Busy Beaver game)**.** *Among all TMs on alphabet $\{0, 1\}$ and with $n = |Q|$ states (not counting the halting one) that halt when run on an empty (i.e., all-zero) tape:*

- *let $\Sigma(n)$ be the largest number of (not necesssarily consecutive) ones left by any machine upon halting;*

- *let $S(n)$ be the largest number of steps performed by any such machine before halting.*

Function $\Sigma(n)$ is known as the *busy beaver* function for n states, and the machine that achieves it is called the Busy Beaver for n states.

Both functions grow very rapidly with n, and their values are only known for $n \leq 4$. The current Busy Beaver candidate with $n = 5$ states writes more than 4K ones before halting after more than 47M steps.

Theorem 8. *The function $S(n)$ is not computable.*

Proof. Suppose that $S(n)$ is computable. Then, we could create a TM to compute $HALT_\varepsilon$ (the variant with empty input) on a machine encoded in string s as follows:

```
on input $s$
    count the number $n$ of states of $\mathcal{M}_s$
    compute $\ell \leftarrow S(n)$
    emulate $\mathcal{M}_s$ for at most $\ell$ steps
    if the emulation halts before $\ell$ steps
        then $\mathcal{M}_s$ clearly halts: accept and halt
        else $\mathcal{M}_s$ takes longer than the BB: reject and halt
```

□

Observe that, by construction, $\Sigma(n) \leq S(n)$ (a TM cannot write more than a symbol per step). The next result is even stronger. Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say that f "eventually outgrows" g, written $f >_E g$, if $f(n) \geq g(n)$ for a sufficiently large value of n:

$$f >_E g \Leftrightarrow \exists N : \forall n > N f(n) \geq g(n).$$

Theorem 9. *The function $\Sigma(n)$ eventually outgrows any computable function.*

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be computable. Let us define the following function:

$$F(n) = \sum_{i=0}^{n} \left[f(i) + i^2 \right].$$

By definition, F clearly has the following properties:

$$F(n) \geq f(n) \quad \forall n \in \mathbb{N}, \tag{1.1}$$

$$F(n) \geq n^2 \quad \forall n \in \mathbb{N}, \tag{1.2}$$

$$F(n+1) > F(n) \quad \forall n \in \mathbb{N} \tag{1.3}$$

the latter because $F(n + 1)$ is equal to $F(n)$ plus a strictly positive term. Moreover, since f is computable, F is computable too. Suppose that M_F is a TM on alphabet $\{0, 1\}$ that, when positioned on the rightmost symbol of an input string of x ones and executed, outputs a string of $F(x)$ ones (i.e., computes the function $x \mapsto F(x)$ in unary representation) and halts below the rightmost one. Let C be the number of states of M_F.

Given an arbitrary integer $x \in \mathbb{N}$, we can define the following machine $\mathcal{M}$ running on an initially empty tape (i.e., a tape filled with zeroes):

- Write x ones on the tape and stop at the rightmost one (i.e., the unary representation of x: it can be done with x states, see Exercise 2 at page 42);
- Execute M_F on the tape (therefore computing $F(x)$ with C states);
- Execute M_F again on the tape (therefore computing $F(F(x))$ with C more states).

The machine $\mathcal{M}$ works on alphabet $\{0,1\}$, starts with an empty tape, ends with $F(F(x))$ ones written on it and has $x+2C$ states; therefore it is a busy beaver candidate, and the $(x+2C)$-state busy beaver must perform at least as well:

$$\Sigma(x+2C) \geq F(F(x)). \tag{1.4}$$

Now,

$$F(x) \geq x^2 >_E x + 2C;$$

the first inequality comes from (1.2), while the second stems from the fact that x^2 eventually dominates any linear function of x. By applying F to both the left- and right-hand sides, which preserves the inequality sign because of (1.3), we get

$$F(F(x)) >_E F(x+2C). \tag{1.5}$$

By concatenating (1.4), (1.5) and (1.1), we get

$$\Sigma(x+2C) \geq F(F(x)) >_E F(x+2C) \geq f(x+2C).$$

Finally, by replaxing $n = x+2C$, we obtain

$$\Sigma(n) >_E f(n).$$

□

This proof is based on the original one by Rado (1962)[9].

1.3.4 Reductions

Note that a few results in the past sections (Theorems 4, 5 and 8) made use of similar arguments: "If A were computable, then we could use it to solve B; however, we know that B is uncomputable, therefore A is too." Now we want to formalize such reasoning scheme.

Definition 5. *Let $L_1 \subset \Sigma_1^*$ and $L_2 \subset \Sigma_2^*$ be two languages (on possibly different alphabets). A function*

$$f : \Sigma_1^* \to \Sigma_2^*$$

is said to be a reduction *from L_1 to L_2 if*

$$\forall s \in \Sigma_1^* \quad s \in L_1 \Leftrightarrow f(s) \in L_2.$$

Basically, we can use a reduction to transform the question "Does s belong to L_1?" into the equivalent question "Does $f(s)$ belong to L_2?"

Clearly, to be useful in computability results, f has to be computable (meaning, as usual, that there is a TM $\mathcal{M}_f$ that computes f).

Definition 6. *We say that $f : \Sigma_1^* \to \Sigma_2^*$ is a* Turing reduction *from $L_1 \subset \Sigma_1^*$ to $L_2 \subset \Sigma_2^*$ if it is a reduction from L_1 to L_2 and it is computable.*

If f is a reduction from L_1 to L_2 we write $L_1 <_f L_2$. In general, if there is a Turing reduction from L_1 to L_2 we say that L_1 is Turing reducible to L_2 and write $L_1 <_T L_2$.

Note that we do *not* require f to have any specific property such as being injective or surjective: just that it "does its work" by transforming any element of L_1 into an element of L_2 and every string that is not in L_1 into a string that is not in L_2.

All computability proofs by reduction follow one of the schemes listed in the following theorem:

[9]See for instance:
`http://computation4cognitivescientists.weebly.com/uploads/6/2/8/3/6283774/rado-on_non-computable_functions.pdf`

Theorem 10. *Let languages L_1 and L_2 and function f be such that $L_1 <_f L_2$; then*

1. *if L_2 is decidable and f is computable, then L_1 is decidable too;*
2. *if L_1 is undecidable and f is computable, then L_2 is undecidable too;*
3. *lf L_1 is undecidable and L_2 is decidable, then f is uncomputable.*

Proof. The first point is proven by showing that, if we have a machine for f and a machine for L_2 we can build a machine for L_1. Let $\mathcal{M}_{L_2}$ be a TM that decides L_2, and let $\mathcal{M}_f$ be a TM that computes f. Then the machine $\mathcal{M}$ that concatenates an execution of $\mathcal{M}_f$ and an execution of $\mathcal{M}_{L_2}$, i.e. computes $\mathcal{M}(s) = \mathcal{M}_{L_2}\big(\mathcal{M}_f(s)\big)$, decides L_1 by definition of f.

The other two points follow by contradiction.

$\square$

In other words, by writing $L_1 <_T L_2$ we mean that L_1 is "less uncomputable" than L_2.

Observe that the proofs of Theorems 4 and 5 follow the second scheme of Theorem 10, while the proof of Theorem 8 follows the third scheme, where the function $S(n)$ is part of the reduction.

Consequences of the Halting Problem incomputability

If $HALT$ were computable, we would be able to settle any mathematical question that can be disproved by a counterexample (on a discrete set), such as the Collatz conjecture, Goldbach's conjecture[10], the non-existence of odd perfect numbers[11]... We would just need to write a machine that systematically search for one such counterexample and halts as soon as it finds one: by feeding this machine as an input to $\mathcal{H}$, we would know whether a counterexample exists at all or not.

More generally, for every proposition P in Mathematical logic we would know whether it is provable or not: just define a machine that, starting from pre-encoded axioms, systematically generates all their consequences (theorems) and halts whenever it generates P. Machine $\mathcal{H}$ would tell us whether P is ever going to be generated or not.

Note that, in all cases described above, we would only receive a "yes/no" answer, not an actual counterexample or a proof.

1.4 Rice's Theorem

Among all questions that we may ask about a Turing machine $\mathcal{M}$, some of them have a *syntactic* nature, i.e., they refer to its actual implementation: "does $\mathcal{M}$ halt within 50 steps?", "Does $\mathcal{M}$ ever reach state q?", "Does $\mathcal{M}$ ever print symbol σ on the tape?"...

Other questions are of a *semantic* type, i.e., they refer to the language accepted by $\mathcal{M}$, with no regards about $\mathcal{M}$'s behavior: "does $\mathcal{M}$ only accept even-length strings?", "Does $\mathcal{M}$ accept any string?", "Does $\mathcal{M}$ accept at least 100 different strings?"...

Definition 7. *A* property *of a TM is a mapping P from TMs to $\{0,1\}$, and we say that $\mathcal{M}$ has property P when $P(M) = 1$.*

Definition 8. *A property is* semantic *if its value is shared by all TMs recognizing the same language: if $L(\mathcal{M}) = L(\mathcal{M}')$, then $P(\mathcal{M}) = P(\mathcal{M}')$.*

[10]Every even number (larger than 2) can be expressed as the sum of two primes, see `https://en.wikipedia.org/wiki/Goldbach%27s_conjecture`

[11]`https://en.wikipedia.org/wiki/Perfect_number`

By extension, we can say that a language S has a property P if the machine that recognizes S has. Finally, we define a property as *trivial* if all TMs have it, or if no TM has it. A property is *non-trivial* if there is at least one machine having it, and one not having it.

The two trivial properties (the one possessed by all TMs and the one posssessed by none) are easy to decide, respectively by the machine that always accepts and by the one that always rejects. On the other hand:

Theorem 11 (Rice's Theorem)**.** *All non-trivial semantic properties of TMs are undecidable.*

Proof. As usual, let's work by contradiction via reduction from the Halting Problem.
Suppose that a non-trivial semantic property P is decidable; this means that there is a TM $\mathcal{M}_P$ that can be run on the encoding of any TM $\mathcal{M}$ and returns 1 if $\mathcal{M}$ has property P, 0 otherwise.
Let us also assume that the empty language $\emptyset$ does not have the property P (otherwise we can work on the complementary property), and that the Turing machine $\mathcal{N}$ has the property P (we can always find $\mathcal{N}$ because P is not trivial).
Given the strings $s, t \in \Sigma^*$, we can then check whether $\mathcal{M}_s(t)$ halts by building the following auxiliary TM $\mathcal{N}'$ that, on input u, works as follows:

- move the input u onto an auxiliary tape for later use, and replace it with t;
- execute $\mathcal{M}_s$ on input t;
- when the simulation halts (which, as we know, might not happen), restore the original input u on the tape by copying it back from the auxiliary tape;
- run $\mathcal{N}$ on the original input u.

The machine $\mathcal{N}'$ we just defined accepts the same language as $\mathcal{N}$ if $\mathcal{M}_s(t)$ halts, otherwise it runs forever, therefore accepting the empty language. Therefore, running our hypothetical decision procedure $\mathcal{M}_P$ on machine $\mathcal{N}'$ we obtain "yes" if $\mathcal{M}_s(t)$ halts (since in this case $L(\mathcal{N}) = L(\mathcal{N}')$) ,and "no" if $\mathcal{M}_s(t)$ doesn't halt (and thus the empty language, which doesn't have the property P, is recognized). □

Observe that we simply use $\mathcal{N}$, which has the property, as a sort of Trojan horse for computation $\mathcal{M}_s(t)$. See also the Wikipedia entry for Rice's Theorem[12].

[12] `https://en.wikipedia.org/wiki/Rice%27s_theorem`

Chapter 2

Some undecidable problems

2.1 Post Correspondence Problem

The following is an example of a problem that, while not immediately related to a computational device, can be proved to be uncomputable[1]:

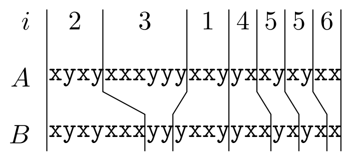
Definition 9 (Post Correspondence Problem — PCP)**.** *Given two sets of n strings, $\{A_1, \dots, A_n\} \subset \Sigma^*$ and $\{B_1, \dots, B_n\} \subset \Sigma^*$, is it possible to find a finite sequence of k indices $1 \le i_1, \dots, i_k \le n$ (in no particular order and possibly with repetitions) such that $A_{i_1} A_{i_2} \cdots A_{i_k} = B_{i_1} B_{i_2} \cdots B_{i_k}$?*

In other words, if $\{(A_1, B_1), (A_2, B_2), \dots, (A_n, B_n)\} \subset \Sigma^* \times \Sigma^*$ is a finite list of *pairs* of strings, is it possible to select a finite sequence of pairs (possibly with repetitions) so that the concatenation of the first members (the A's) is equal to the concatenation of the B's?

As a trivial example, if for a specific index j $A_j = B_j$, then the positive answer to PCP is just the sequence of length $k = 1$ where $i_1 = j$. Another example with a positive answer is the following:

i	A_i	B_i
1	xxy	yxxy
2	xyxy	xyxyxxx
3	xxxyyy	yy
4	yx	yxx
5	xy	yx
6	xx	x

A solution is the index sequence $2, 3, 1, 4, 5, 5, 6$, giving the following concatenations:



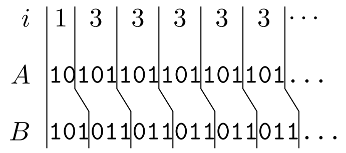
Note that this is actually the concatenation of two simpler solutions: $2, 3, 1$ and $4, 5, 5, 6$.

Some problems have no solution. For instance:

i	A_i	B_i
1	10	101
2	011	11
3	101	011

[1]See the Wikipedia article
`https://en.wikipedia.org/wiki/Post_correspondence_problem`
and also
`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.721.2199&rep=rep1&type=pdf`

The sequence must start with $i_1 = 1$, but then the only way to proceed is to keep concatenating (A_3, B_3), but this way the B sequence is always 1 symbol longer than A:



i | 1 | 3 | 3 | 3 | 3 | 3 | ···
A | 10 | 101 | 101 | 101 | 101 | 101 | ...
B | 101 | 011 | 011 | 011 | 011 | 011 | ...

Let us consider another, simpler variant of the PCP:

Definition 10 (Modified Post Correspondence Problem — MPCP)**.** *In the same conditons of PCP, we furthermore require that the first chosen index is* $i_1 = 1$ *(i.e., pair 1 is initially laid out).*

2.1.1 Undecidability of the Modified PCP

Let us consider a TM $\mathcal{M}$ with the following limitations:

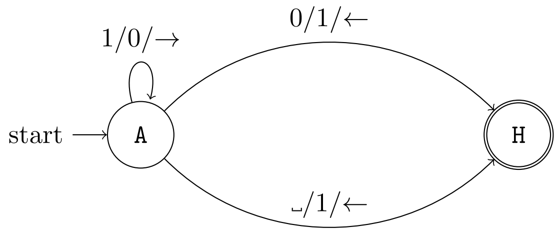
- $\mathcal{M}$ has a 3-symbol alphabet $\Sigma = \{␣, 0, 1\}$, where the default symbol is ␣;
- $\mathcal{M}$ never moves left of its starting position (i.e., the tape only extends indefinitely to the right);
- $\mathcal{M}$ never writes a ␣ (however it still has two symbols to write).

As we have seen, none of these limitations actually impair the universality of $\mathcal{M}$.

A small example

As an example, consider the following 1-state TM $\mathcal{M}$ that increments a binary number whose LSB is at the starting position (`A` is the state name, `H` is the halting state):

	␣	0	1
`A`	1/←/`H`	1/←/`H`	0/→/`A`



We use letters for states in place of the more customary $q_0, q_1, \ldots$ or descriptive names like `start`, `change` because we will need to represent them as symbols in an MPCP instance.

We want to build a Modified PCP instance in which individual strings represent "pieces" of the TM's configuration, while the (A_i, B_i) string pairs "force" the construction of the solution in a way that represents the evolution of the machine's configuration from one step to the other. We will use the following alphabet:

$$\Sigma = \{␣, \texttt{0}, \texttt{1}, \texttt{A}, \texttt{H}, \texttt{\#}, \texttt{\$}\},$$

i.e., all symbols in $\mathcal{M}$'s alphabet, one symbol per state including the halting one, and two separator symbols, "`#`" to separate subsequent steps of $\mathcal{M}$'s execution, and "`$`" to represent the end of the execution.

Remember that a "configuration" of a TM consists of three pieces of information:

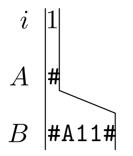
- the content of the tape,

- the current position, and
- the current state.

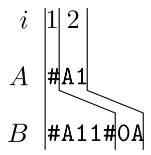
Suppose that in its initial configuration $\mathcal{M}$'s tape contains the string `11`, then its representation will be "`#A11#`", i.e., the tape's content with the state's symbol to the left of the current position, and delimiters`#` to enclose it. Since the first steps involves replacing the leftmost `1` on the tape with a `0` and moving right, the representation of $\mathcal{M}$'s evolution will be "`#A11#0A1#`".

We want to design the Modified PCP instance so that, every time we need to choose a string pair, the choice is (almost) forced, and in a way that the concatenation of the B_i's is always one $\mathcal{M}$'s step further than the concatenation of the A_i's.

We start by forcing the initial pair $A_1 =$ `#`, $B_1 =$ `#A11#`:



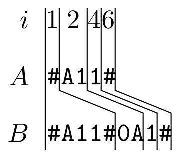
Note that the next character to match in B is a state name, followed by a symbol. Since our transition rule requires the machine to replace the symbol and move right, whenever we find the string "`A1`" we know that the next configuration will need to contain "`0A`". That will be our second string pair ($A_2 =$ `A1`, $B_2 =$ `0A`), and we will be able to proceed with the string composition as follows:



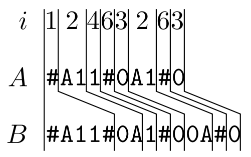
In order to complete the first step, all other symbols that are not in the current position must be copied, therefore we will need a bunch of other "copying" rules (one per tape symbol, one for the state separator)

$$A_3 = B_3 = \texttt{0}, \quad A_4 = B_4 = \texttt{1}, \quad A_5 = B_5 = \llcorner\!\lrcorner, \quad A_6 = B_6 = \texttt{\#}.$$

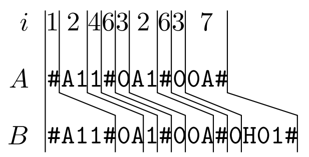
Note that these rules would make the original PCP trivial, but we are working with the modified version where an initial string is forced. With these new rules we can advance the matching:



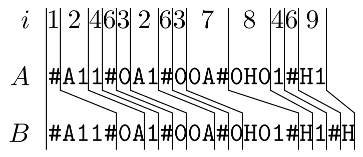
Note that the existing rules allow us to take the matching still further:



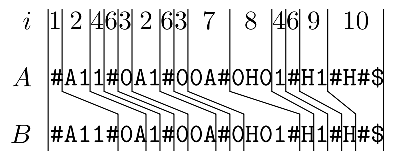
Note that in the configuration that we are currently trying to match the state letter is on the right of all tape symbols. This means that the current symbol is a blank, and $\mathcal{M}$'s transition rule requires to write "`1`", move left and halt. Since we need to move left, we cannot use the pair $i = 3$ to proceed, because the next symbol to appear in B should be the state letter "`H`"; to move left, we introduce the pair $A_7 =$ `0A#`, $B_7 =$ `H01#`, and the matching becomes:



Now string B represents the whole execution of $\mathcal{M}$ on input "`11`". Still, $A \neq B$. We will now introduce a few ad-hoc steps that, upon reaching the halting state H, get rid of all tape symbols and keep the state as the only useful information. Whenever we need to match anything in the form "σH" or "$H\sigma$", where σ is a tape symbol, we can proceed by leaving only "`H`" on B. We can add a shortcut by also matching any string in the form "$\sigma_1 H \sigma_2$". In this case, the two following pairs will do the trick: $A_8 = \texttt{0H0}$, $B_8 = \texttt{H}$ and $A_9 = \texttt{H1}$, $B_9 = \texttt{H}$.



Note how, as we clean out tape symbols, string A starts catching up to B. When B's configuration is reduced to just the Halting state symbol, we can finally close the matching by adding the following final pair to the instance, where we use the finalization marker "\$": $A_{10} = \texttt{\#H\#\$}$, $B_{10} = \texttt{\#\$}$.


i 1 2 4 6 3 2 6 3 7 8 4 6 9 10
A #A11#0A1#00A#0H01#H1#H#$
B #A11#0A1#00A#0H01#H1#H#$

We have therefore constructed a Modified PCP instance that mimicks the evolution of $\mathcal{M}$ and that has a solution precisely because the machine halts.

General case

Based on the previous example, consider a TM $\mathcal{M}$ on an alphabet $\Sigma_{\mathcal{M}}$ and stateset Q, including the halting states, with the limitations discussed above. Given the initial tape content (input) $x \in \Sigma^*_{\mathcal{M}}$, we can simulate the machine's execution by building a MPCP instance on the alphabet

$$\Sigma = \Sigma_{\mathcal{M}} \cup Q \cup \{\texttt{\#}, \texttt{\$}\}$$

with the following string pairs:

- Initial pair: $A_1 = \texttt{\#}$, $B_1 = \texttt{\#A}x\texttt{\#}$.
  B_1 represents the machine in its initial configuration. With the rules below, any attempt to match syting B as it grows will result in following $\mathcal{M}$'s evolution past the initial configuration.
- Copy pairs: for every symbol $\sigma \in \Sigma_{\mathcal{M}}$, add pair $A_i = B_i = \sigma$. Also add the pair $A_i = B_i = \texttt{\#}$ to propagate the "end of step" symbol.
  These pairs are needed to propagate the symbols on the tape outside the current position, in the sense that every time we add one such A_i to extend string A, the same symbol will be added to B by means of the corresponding B_i.
- Rule pairs: Add string pairs that represent the transition rules at the current position:

For every state of the form:	Add the following string pairs:
$(\sigma, S) \mapsto (\sigma', S', \rightarrow)$	$A_i = S\sigma$, $B_i = \sigma' S'$
$(\sigma, S) \mapsto (\sigma', S', \leftarrow)$	$A_i = \mu S \sigma$, $B_i = S' \mu \sigma'$ for every $\mu \in \Sigma_{\mathcal{M}}$
	(if $\sigma = ␣$, then add a pair with $\sigma = \texttt{\#}$).

These pairs are the only

ones such that a non-halting state appears in a string A_i. Therefore, in order to extend the matching we will be forced to use them whenever a non-halting state symbol appears in B, enforcing the application of the transition function to the next step.
Note that in the initial pair $|A_1| < |B_1|$, and that for all other pairs listed up to now $|A_i| \leq |B_i|$; therefore, string B will always be longer than string A.

- Final cleanup: for all halting states $H \in Q$ and all tape symbols $\sigma, \sigma' \in \Sigma_{\mathcal{M}}$ add the following string pairs:

$$A_i = \sigma H, B_i = H; \qquad A_i = H\sigma, B_i = H; \qquad A_i = \sigma H \sigma', B_i = H.$$

  As said before, these pairs apply to the halting state and "consume" all tape symbols appearing in B until the halting state alone appears. Note that these are the only pairs up to now where $|A_i| > |B_i|$; therefore, until a halting state appears, there is no hope to get string A as long as string B.

- Closing pair: for all halting states H, add pair $A_i = \#H\#\$$, $B_i = \$$.
  This puts an end to the matching rush: string A is matched to the remaining part of string B.

By the considerations about matching and string lengths, one should remain convinced that the MPCP with the proposed set of string pairs has a solution if and only if $\mathcal{M}(x)$ halts, therefore proving the following theorem:

Theorem 12. *The Modified Post Correspondence Problem is undecidable.*

2.1.2 Undecidability of the Post Correspondence Problem

So far, we have been considering the "modified" case in which an initial pair is enforced. Now we need a way to transform an instance of the MPCP into an equivalent instance (i.e., with the same solution or lack thereof) of the PCP.

Suppose that the n pairs $(A_1, B_1), (A_2, B_2), \dots, (A_n, B_n)$ are an instance of the Modified PCP.

We want to transform it into a PCP instance (i.e., an instance that does not explicitly require the first chosen pair to be $i = 1$). Let $*$ be a symbol not present in the strings. Then we can create the pairs (A'_i, B'_i) by putting a "*" *before* every symbol in A_i and *after* every symbol in B_i. So far, the PCP would have no solution: all strings in A start with the new symbol, while no string in B does.

In order to enforce the first pair, let us introduce the new pair (A_0, B_0) where $A_0 = A_1$ and $B_0 = *B_1$. Being (so far) the only pair starting with the same symbol, (A_0, B_0) is the only viable first choice.

Let $1, i_2, i_3, \dots, i_k$ be a solution to the original MPCP, i.e., $A_1 A_{i_2} \cdots A_{i_k} = B_1 B_{i_2} \cdots B_{i_k}$. Then, the sequence of indices $0, i_2, \dots, i_k$ is *almost* a solution to the PCP problem that we are trying to build, in the sense that $B'_0 B'_{i_2} \cdots B'_{i_k}$ is one "*" longer than $A'_0 A'_{i_2} \cdots A'_{i_k}$. Therefore we add one last pair (A'_{n+1}, B'_{n+1}) to "absorb" the asterisk: $A'_{n+1} = *\$$, $B'_{n+1} = \$$.

As an example, here is a conversion from a MPCP instance to an equivalent PCP instance:

MPCP

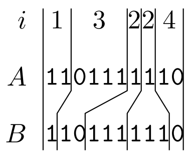
i	A_i	B_i
1	11	1
2	1	111
3	0111	10
4	10	0

$\Rightarrow$

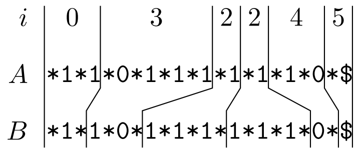
PCP

i	A_i	B_i
0	*1*1	*1*
1	*1*1	1*
2	*1	1*1*1*
3	*0*1*1*1	1*0*
4	*1*0	0*
5	*$	$

A solution to the MPCP is the $i_1 = 1$, $i_2 = 3$, $i_3 = 2$, $i_4 = 2$, $i_5 = 4$:



The corresponding solution to the PCP problem is $i_1 = 1$, $i_2 = 3$, $i_3 = 2$, $i_4 = 2$, $i_5 = 4$, $i_6 = 5$:



The above described construction provides a PCP instance that is solvable if and only if the original MPCP instance was solvable. In addition, the construction is clearly computable and is therefore a Turing reduction from MPCP to PCP.

This proves the following

Theorem 13. *The Post correspondence problem is uncomputable.*

2.2 Kolmogorov complexity

We are quite used to programs that "compress" our files in order to save space on our mass storage media. Programs such as WinZip, WinRAR, gzip, bzip2, xzip, 7z basically operate by identifying predictable patterns in the sequence of symbols that compose the original file and replacing them with shorter descriptions according to a predefined language.

Since a file is just a string of symbols, we can ask ourselves "how much can a given string be compressed?"

To better formalize the question, let us consider the following setting:

Definition 11. *Let Σ be a suitable alphabet (e.g., ASCII or Unicode), let $\mathcal{U}$ be a universal turing machine working on Σ and let $x \in \Sigma^*$ be a string. We say that the pair of strings $D = (s,t) \in \Sigma^* \times \Sigma^*$ is a* description *of x if s encodes a TM which, when simulated by $\mathcal{U}$ on input t, produces x on the tape and halts:*

$$\mathcal{U}(s,t) = \mathcal{M}_s(t) = x.$$

In other words, we are formalizing in terms of Turing machines a very common scenario: $\mathcal{U}$ is our computer (with its operating system), while $D = (s,t)$ is a *self-extracting* executable file where s is the code that performs the decompression and t is the actual string being decompressed. We usually "run" the decompression code by double-clicking on its icon.

Another way of looking at the definition is to think of $\mathcal{U}$ as a programming language, s as a program written in that language and t as its input.

Of course, every string has many possible descriptions.

We are interested, once $\mathcal{U}$ is fixed, in finding the "most compressed" description for a string x.

Definition 12. *Given a UTM $\mathcal{U}$ and a string x, we define its* Kolmogorov complexity[2] *$K_\mathcal{U}(x)$ to be the size of its smallest description in $\mathcal{U}$:*

$$K_\mathcal{U}(x) = \min\{|(s,t)| : \mathcal{U}(s,t) = x\}.$$

We assume that $|(s,t)| = |s| + |t|$ (i.e., the size of the description is the size of the input string t plus the size of the decompressing program s).

[2]See the Wikipedia article
`https://en.wikipedia.org/wiki/Kolmogorov_complexity`

2.2.1 Dependence on the underlying computational model

Note that the definition of Kolmogorov complexity depends on the chosen computational substrate (the UTM $\mathcal{U}$). Different machines have different encodings, with different sizes, in the same way that different languages can express the same algorithm in more or less concise ways.

Theorem 14. *Given two UTMs $\mathcal{U}$ and $\mathcal{V}$, there is a* constant *value $c_{\mathcal{U}\mathcal{V}}$ such that, for every x,*

$$|K_{\mathcal{U}}(x) - K_{\mathcal{V}}(x)| \leq c_{\mathcal{U}\mathcal{V}}.$$

Note that the constant is independent of the specific string x.

Proof. Let $x \in \Sigma^*$.
Let $D_{\mathcal{U}} = (s_{\mathcal{U}}, t_{\mathcal{U}})$ be a shortest description of x in $\mathcal{U}$ (i.e., such that $\mathcal{U}(D_{\mathcal{U}}) = x$ and $|D_{\mathcal{U}}| = K_{\mathcal{U}}(x)$).
Conversely, let $D_{\mathcal{V}} = (s_{\mathcal{V}}, t_{\mathcal{V}})$ be a shortest description of x in $\mathcal{V}$ (i.e., such that $\mathcal{V}(D_{\mathcal{V}}) = x$ and $|D_{\mathcal{V}}| = K_{\mathcal{V}}(x)$).
Since $\mathcal{U}$ is a UTM, it can be used to simulate $\mathcal{V}$. Let v be the representation of $\mathcal{V}$ in $\mathcal{U}$. Therefore, $(v, D_{\mathcal{V}})$ is a description of x in $\mathcal{U}$. In fact,

$$\mathcal{U}(v, D_{\mathcal{V}}) = \mathcal{V}(D_{\mathcal{V}}) = x.$$

Therefore, $|(v, D_{\mathcal{V}})| \geq K_{\mathcal{U}}(x)$, and thus

$$K_{\mathcal{U}}(x) - K_{\mathcal{V}}(x) \leq |v|.$$

By exchanging $\mathcal{U}$ and $\mathcal{V}$, let u be an encoding of $\mathcal{U}$ that allows us to simulate it with $\mathcal{V}$; we obtain the symmetric inequality:

$$K_{\mathcal{V}}(x) - K_{\mathcal{U}}(x) \leq |u|.$$

By combining the two constants, $c_{\mathcal{U}\mathcal{V}} = \max\{u, v\}$, we obtain the thesis. □

The theorem tells us that the specific computing substrate is not very influent, as the size of x grows, because the difference is constant.

This corresponds to having a self-extracting executable created for a specific OS (say Windows) and asking if a similar compression level would be achievable on Windows. The answer is yes because, given any Linux executable of any size, we can transform it into a Windows executable by prepending to it a Linux simulator for Windows: with a fixed overhead (the Linux simulator for Windows), every self-extracting file for Linux becomes a valid self-extracting file for Windows.

2.2.2 Uncomputability of Kolmogorov complexity

However, we can prove that we cannot compute the Kolmogorov complexity of a generic string. In other words, we cannot be sure that a given description is the most compressed.

Theorem 15. *Given the UTM $\mathcal{U}$, the function $K_{\mathcal{U}} : \Sigma^* \to \mathbb{N}$ is uncomputable.*

Proof. By contradiction, suppose that $\mathcal{M}$ is a TM that computes $K_{\mathcal{U}}$. Suppose that $\mathcal{M}$ is represented by string m in $\mathcal{U}$.
Let us create the following Turing machine $\mathcal{N}$:

```
for all $s \in \Sigma^*$
    if $\mathcal{M}(s) \geq |m| + 1000000$
        write $s$
        halt
```

Observe the following:

- $\mathcal{N}$ does not take an input, and outputs a string whose Kolmogorov complexity (wrt $\mathcal{U}$) is greater or equal to $|m| + 1000000$.
- $\mathcal{N}$ contains $\mathcal{M}$ as a "subroutine", but we can safely assume that its description does not add more than a million symbols to that of $\mathcal{M}$.

Let x be the string written by $\mathcal{N}$ starting from the empty input. From the first point, we know that

$$K_{\mathcal{U}}(x) \geq |m| + 1000000.$$

On the other hand, let n be the string that represents $\mathcal{N}$; from the second point we know that (n, ε) is a description of x, therefore

$$K_{\mathcal{U}}(x) \leq |n| < |m| + 1000000,$$

therefore we have a contradiction. Observe that, if 1000000 looks too small an overhead, we can increase it as much as we want. □

We have searched for a string x of high Kolmogorov complexity, and in the process we have been able to generate it with a machine whose size is smaller than the (alleged) complexity of x.

This theorem is a formal rendition of the famous Berry paradox:

"The smallest positive integer not definable with less than thirteen Englishwords."

defines such an integer with twelve words.

Chapter 3

Complexity classes: P and NP

From now on, we will be only dealing with computable functions; the algorithms that we will analize will always terminate, and our main concern will be about the amount of resources (time, space) required to compute them.

3.1 Definitions

When discussing complexity, we are mainly interested in the relationship between the size of the input and the execution "time" of an algorithm executed by a Turing machine. We still refer to TMs because both input size and execution time can be defined unambiguously in that model.

Input size

By "size" of the input, we mean the number of symbols used to encode it in the machine's tape. Since we are only concerned in asymptotic relationships, the particular alphabet used by a machine is of no concern, and we may as well just consider machines with alphabet $\Sigma = \{0, 1\}$.
We require that the input data are encoded in a reasonable way. For instance, numbers may be represented in base-2 notation (although the precise base does not matter when doing asymptotic analysis), so that the size of the representation $r_2(n)$ of integer n in base 2 is logarithmic with respect to its value:

$$|r_2(n)| = O(\log n).$$

In this sense, unary representations (representing n by a string of n consecutive 1's) is not to be considered reasonable because its size is exponential with respect to the base-2 notation.

Execution time

We dub "execution time," or simply "time," the number of steps required by a TM to get to a halting state. Let $\mathcal{M}$ be a TM that always halts. We can define the "time" function

$$\begin{aligned} t_{\mathcal{M}} : \Sigma^* &\rightarrow \mathbb{N} \\ x &\mapsto \text{\# of steps before } \mathcal{M} \text{ halts on input } x \end{aligned}$$

that maps every input string x onto the number of steps that $\mathcal{M}$ performs upon input x before halting. $\mathcal{M}$ always halts, so it is a well-defined function. Since the number of strings of a given size n is finite, we can also define (and actually compute, if needed) the following "worst-case" time for inputs of size n:

$$\begin{aligned} T_{\mathcal{M}} : \mathbb{N} &\rightarrow \mathbb{N} \\ n &\mapsto \max\{t_{\mathcal{M}}(x) : x \in \Sigma^n\}, \end{aligned}$$

i.e., $T_{\mathcal{M}}(n)$ is the longest time that $\mathcal{M}$ takes before halting on an input of size n.

3.2 Polynomial languages

Let us now focus on decision problems.

Definition 13. *Let $f : \mathbb{N} \to \mathbb{N}$ be any computable function. We say that a language $L \subseteq \Sigma^*$ is of class DTIME(f), and write $L \in$ DTIME(f), if there is a TM $\mathcal{M}$ that decides L and its worst-case time, as a function of input size, is dominated by f:*

$$L \in \mathit{DTIME}(f) \quad \Leftrightarrow \quad \exists \mathcal{M}: \quad L(\mathcal{M}) = L \land T_{\mathcal{M}} = O(f).$$

In other words, DTIME(f) is the class of all languages that can be decided by some TM in time eventually bounded by function $c \cdot f$, where c is constant.
Saying $L \in$ DTIME(f) means that there is a machine $\mathcal{M}$, a constant $c \in \mathbb{N}$ and an input size $n_0 \in \mathbb{N}$ such that, for every input x with size larger than n_0, $\mathcal{M}$ decides $x \in L$ in at most $c \cdot f(|x|)$ steps.

Languages that can be decided in a time that is polynomial with respect to the input size are very important, so we give a short name to their class:

Definition 14.

$$\boldsymbol{P} = \bigcup_{k=0}^{\infty} \mathit{DTIME}(n^k).$$

In other words, we say that a language $L \in \Sigma^*$ is polynomial-time*, and write $L \in \boldsymbol{P}$, if there are a machine $\mathcal{M}$ and a polynomial $p(n)$ such that for every input string x*

$$x \in L \quad \Leftrightarrow \quad \mathcal{M}(x) = 1 \land t_{\mathcal{M}}(x) \leq p(|x|). \tag{3.1}$$

3.2.1 Examples

Here are some examples of polynomial-time languages.

CONNECTED — Given an encoding of graph G (e.g., the number of nodes followed by an adjacency matrix or list), $G \in$ CONNECTED if and only if there is a path in G between every pair of nodes.

PRIME — Given a base-2 representation of a natural number N, we say that $N \in$ PRIME if and only if N is, of course, prime.
Observe that the naive algorithm "divide by all integers from 2 to $\lfloor\sqrt{N}\rfloor$" is *not* polynomial with respect to the size of the input string. In fact, the input size is $n = O(\log N)$ (the number of bits used to represent a number is logarithmic with respect to its magnitude), therefore the naive algorithm would take $\lfloor\sqrt{N}\rfloor - 1 = O(2^{n/2})$ divisions in the worst case, which grows faster than any polynomial[1].
Anyway, it has recently been shown[2] that PRIME $\in$ **P**.

(Counter?)-examples

On the other hand, we do not know of any polynomial-time algorithm for the following languages:

SATISFIABILITY or SAT — Given a Boolean expression $f(x_1, \dots, x_n)$ (usually in conjunctive normal form, CNF[3]) involving n variables, is there a truth assignment to the variables that satisfies (i.e., makes true) the formula[4]?

[1]An algorithm that is polynomial with respect to the *magnitude* of the numbers instead than the size of their representation is said to be "pseudo-polynomial." In fact, the naive primality test would be polynomial if we chose to represent N in unary notation (N consecutive 1's).

[2]`https://en.wikipedia.org/wiki/Primality_test#Fast_deterministic_tests`

[3]`https://en.wikipedia.org/wiki/Conjunctive_normal_form`

[4]`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

CLIQUE — Given an encoding of graph G and a number k, does G contain k nodes that are all connected to each other[5]?

TRAVELING SALESMAN PROBLEM or TSP — Given an encoding of a complete *weighted* graph G (i.e., all pairs of nodes are connected, and pair i, j is assigned a "weight" w_{ij}) and a "budget" k, is there an order of visit (permutation) σ of all nodes such that

$$\left(\sum_{i=1}^{n-1} w_{\sigma_i\sigma_{i+1}}\right) + w_{\sigma_n\sigma_1} \leq k, \tag{3.2}$$

i.e., the total weight along the corresponding closed path in that order of visit (also considering return to the starting node) is within budget[6]?

However, we have no proof that these languages (and many others) are not in **P**. In the following section, we will try to characterize these languages.

3.2.2 Example: Boolean formulas and the conjunctive normal form

To clarify the SAT example, let us specify how a typical SAT instance is represented.

Given n boolean variables $x_1, \dots, x_n$, we can define the following:

- a *term*, or *literal*, is a variable x_i or its negation $\neg x_i$;
- a *clause* is a disjunction of terms;
- finally, a *formula* or *expression* is a conjunction of clauses.

Definition 15 (Conjunctive Normal Form (CNF))**.** *A formula f is said to be in* conjunctive normal form *with n variables and m clauses if it can be written as*

$$f(x_1, \dots, x_n) = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{l_i} g_{ij},$$

where clause i has l_i terms, every literal g_{ij} is in the form x_k or in the form $\neg x_k$.

For instance, the following is a CNF formula with $n = 5$ variables and $m = 4$ clauses:

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) \quad = \quad & (x_1 \vee \neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \\ & \wedge(\neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5). \end{aligned} \tag{3.3}$$

Asking about the satisfiability of a CNF formula f amount at asking for a truth assignment such that every clause has at least one true literal. For example, the following assignment, among many others, satisfies (3.3):

$$x1 = x2 = \text{true}; \quad x_3 = x_4 = x_5 = \text{false}.$$

We can therefore say that $f \in \text{SAT}$.

Note that CNF is powerful enough to express any (unquantified) statement about boolean variables. For instance, the following 2-variable formula is satisfiable only by variables having the same truth value:

$$(\neg x \vee y) \wedge (x \vee \neg y).$$

It therefore "captures" the idea of equality in the sense that it is true whenever $x = y$. In fact, the clause $(\neg x \vee y)$ means "x implies y."

[5]`https://en.wikipedia.org/wiki/Clique_(graph_theory)`

[6]`https://en.wikipedia.org/wiki/Travelling_salesman_problem`

Moreover, there are standard ways to convert *any* Boolean formula to CNF, based on some simple transformation rules, easily verifiable by testing all possible combinations of values — or just by reasoning:

$$\begin{aligned}
a \vee (b \wedge c) &\equiv (a \vee b) \wedge (a \vee c) \\
a \wedge (b \vee c) &\equiv (a \wedge b) \vee (a \wedge c) \\
\neg(a \vee b) &\equiv \neg a \wedge \neg b \\
\neg(a \wedge b) &\equiv \neg a \vee \neg b \\
a \rightarrow b &\equiv \neg a \vee b.
\end{aligned}$$

3.3 NP languages

While the three languages listed above (SAT, CLIQUE, TSP) cannot be decided by any known polynomial algorithm, they share a common property: if a string is in the language, there is an "easily" (polynomially) verifiable proof of it:

- If $f(x_1, \dots, x_n) \in \text{SAT}$ (i.e., boolean formula f is satisfiable), then there is a truth assignment to the variables $x_1, \dots, x_n$ that satisfies it. If we were given this truth assignment, we could easily check that, indeed, $f \in \text{SAT}$. Note that the truth assignment consists of n truth values (bits) and is therefore shorter than the encoding of f (which contains a whole boolean expression on n variables), and that computing a Boolean formula can be reduced to a finite number of scans.
- If $G \in \text{CLIQUE}$, then there is a list of k interconnected nodes; given that list, we could easily verify that G contains all edges between them. The list contains k integers from 1 to the number of nodes in G (which is polynomial with respect to the size of G's representation) and requires a presumably quadratic or cubic time to be checked.
- If $G \in \text{TSP}$, then there is a permutation of the nodes in G, i.e., a list of nodes. Given that list, we can easily sum the weights as in (3.2) and check that the inequality holds.

In other words, if we are provided a *certificate* (or *witness*), it is easy for us to check that a given string belongs to the language. What's important is that both the certificate's size and the time to check are polynomial with respect to the input size. The class of such problems is called **NP**. More formally:

Definition 16. *We say that a language $L \subseteq \Sigma^*$ is of class **NP**, and write $L \in$ **NP**, if there is a TM $\mathcal{M}$ and two polynomials $p(n)$ and $q(n)$ such that for every input string x*

$$x \in L \quad \Leftrightarrow \quad \exists c \in \Sigma^{q(|x|)} : \mathcal{M}(x, c) = 1 \wedge t_{\mathcal{M}}(x, c) \le p(|x|). \tag{3.4}$$

Basically, the two polynomials are needed to bound both the size of certificate c and the execution time of $\mathcal{M}$.

Observe that the definition only requires a (polynomially verifiable) certificate to exist only for "yes" answers, while "no" instances (i.e., strings x such that $x \notin L$) might not be verifiable.

3.3.1 Non-deterministic Turing Machines

An alternative definition of **NP** highlights the meaning of the class name, and will be very useful in the future.

Definition 17. *A non-deterministic Turing Machine (NDTM) is a TM with two different, independent transition functions. At each step, the NDTM makes an arbitrary choice as to which function to apply. Every sequence of choices defines a possible* computation *of the NDTM. We say that the NDTM* accepts *an input x if at least one computation (i.e., one of the possible arbitrary sequences of choices) terminates in an accepting state.*

There are many different ways of imagining a NDTM: one that flips a coin at each step, one that always makes the right choice towards acceptance, one that "doubles" at each step following both choices at once. Note that, while a normal, deterministic TM is a viable computational model, a NDTM is not, and has no correspondence to any current or envisionable computational device[7].

Alternate definitions might refer to machines with more than two choices, with a subset of choices for every input, and so on, but they are all functionally equivalent.

We can define the class NTIME(f) as the NDTM equivalent of class DTIME(f), just by replacing the TM in Definition 13 with a NDTM:

Definition 18. *Let $f : \mathbb{N} \to \mathbb{N}$ be any computable function. We say that a language $L \subseteq \Sigma^*$ is of class NTIME(f), and write $L \in$ NTIME(f), if there is a NDTM $\mathcal{M}$ that decides L and its worst-case time, as a function of input size, is dominated by f:*

$$L \in \textit{NTIME}(f) \quad \Leftrightarrow \quad \exists \quad \textit{NDTM} \quad \mathcal{N}: \quad L(\mathcal{N}) = L \wedge T_{\mathcal{N}} = O(f).$$

Indeed, the names "DTIME" and "NTIME" refer to the deterministic and non-deterministic reference machine. Also, the name **NP** means "non-deterministically polynomial (time)," as the following theorem implies by setting a clear parallel between the definition of **P** and **NP**:

Theorem 16.

$$\textbf{\textit{NP}} = \bigcup_{k=0}^{\infty} \textit{NTIME}(n^k).$$

Proof. See also Theorem 2.6 in the online draft of Arora-Barak. We can prove the two inclusione separately.

Let $L \in$ **NP**, as in Definition 16. We can define a NDTM $\mathcal{N}$ that, given input x, starts by non-deterministically appending a certificate $c \in \Sigma^{q(|x|)}$: every computation generates a different certificate. After this non-deterministic part, we run the machine $\mathcal{M}$ from Definition 16 on the tape containing (x, c). If $x \in L$, then at least one computation has written the correct certificate, and thus ends in an accepting state. On the other hand, if $x \not\in L$ then no certificate can end in acceptance. Therefore, $\mathcal{N}$ accepts x if and only if $x \in L$. The NDTM $\mathcal{N}$ performs $q(|x|)$ steps to write the (non-deterministic) certificate, followed by the $p(|x|)$ steps due to the execution of $\mathcal{M}$, and is therefore polynomial with respect to the input. Thus, $L \in \text{NTIME}(n^k)$ for some $k \in \mathbb{N}$.

Conversely, let $L \in \text{NTIME}(n^k)$ for some $k \in \mathbb{N}$. This means that x can be decided by a NDTM $\mathcal{N}$ in time $q(|x|) = O(|x|^k)$, during which it performs $q(|x|)$ arbitrary binary choices. Suppose that $x \in L$, then there is an accepting computation by $\mathcal{N}$. Let $c \in \{0,1\}^{q(|x|)}$ be the sequence of arbitrary choices done by the accepting computation of $\mathcal{N}(x)$. We can use c as a certificate in Definition 16, by creating a deterministic TM $\mathcal{M}$ that uses c to emulate $\mathcal{N}(x)$'s accepting computation by performing the correct choices at every step. If $x \not\in L$, then no computation by $\mathcal{N}(x)$ ends by accepting the input, therefore all certificates fail, and $\mathcal{M}(x, c) = 0$ for every c. Thus, all conditions in Definition 16 hold, and $L \in$ **NP**. □

3.4 Reductions and hardness

Nobody knows if **NP** is a proper superset of **P**, yet. In order to better assess the problem, we need to set up a hierarchy within **NP** in order to identify, if possible, languages that are harder than others. To do this, we resort again to *reductions*.

Definition 19. *Given two languages $L, L' \in$ **NP**, we say that L is* polynomially reducible *to L', and we write $L \leq_p L'$, if there is a function $R : \Sigma^* \to \Sigma^*$ such that*

$$x \in L \quad \Leftrightarrow \quad R(x) \in L'$$

and R halts in polynomial time wrt $|x|$.

[7]Not even quantum computing, no matter what popular science magazines write.

In other words, R maps strings in L to strings in L' and strings that are not in L to strings that are not in L'. Note that we require R to be computable in polynomial time, i.e., there must be a polynomial $p(n)$ such that $R(x)$ is computed in at most $p(|x|)$ steps. If $L \leq_p L'$, we say that L' is at least as hard as L. In fact, if we have a procedure to decide L', we can apply it to decide also L with "just" a polynomial overhead due to the reduction.

3.4.1 Simple examples

Reductions between versions of SAT

Definition 20 (k-CNF)**.** *If all clauses of a CNF formula have at most k literals in them, then we say that the formula is k-CNF (conjunctive normal form with k-literal clauses).*

For instance, (3.3) is 4-CNF and, in general, k-CNF for all $k \geq 4$. It is not 3-CNF because it has some 4-literal clauses. Sometimes, the definition of k-CNF is stricter, and requires that every clause has *precisely* k literals. Nothing changes, since we can always write the same literal twice in order to fill the clause up.

Definition 21. *Given $k \in \mathbb{N}$, the language k-SAT is the set of all (encodings of) satisfiable $k-CNF$ formulas.*

Let us start with a "trivial" theorem:

Theorem 17. *Given $k \in \mathbb{N}$,*

$$k\text{-}SAT \leq_p SAT.$$

Proof. Define the reduction $R(x)$ as follows: given a string x, if it encodes a k-CNF formula, then leave it as it is; otherwise, return an unsatisfiable formula. $\square$

The simple reduction takes into account the fact that k-SAT $\subseteq$ SAT, therefore if we are able to decide SAT, we can a fortiori decide k-SAT.

The following fact is less obvious:

Theorem 18.

$$SAT \leq_p 3\text{-}SAT.$$

Proof. Let f be a CNF formula. Suppose that f is not 3-CNF. Let clause i have $l_i > 3$ literals:

$$\bigvee_{j=1}^{l_i} g_{ij} \tag{3.5}$$

Let us introduce a new variable, h, and split the clause as follows,

$$\left(h \vee \bigvee_{j=1}^{l_i-2} g_{ij} \right) \wedge (\neg h \vee g_{i,l_i-1} \vee g_{il_i}), \tag{3.6}$$

by keeping all literals, apart from the last two, in the first clause, and putting the last two in the second one. By construction, the truth assignments that satisfy (3.5) also satisfy (3.6), and viceversa. In fact, if (3.5) is satisfied then at least one of its literals are true; but then one of the two clauses of (3.6) is satisfied by the same literal, while the other can be satisfied by appropriately setting the value of the new variable h. Conversely, if both clauses in (3.6) are satisfied, then at least one of the literals in (3.5) is true, because the truth value of h alone cannot satisfy both clauses.

The step we just described transforms an l_i-literal clause into the conjunction of an $(l_i - 1)$-literal clause and a 3-literal clause which is satisfiable if and only if the original one was; by applying it recursively, we end up with a 3-CNF formula which is satisfiable if and only if the original f was. $\square$

As an example, the 4-CNF formula (3.3) can be reduced to the following 3-CNF with the two additional variables h and k used to split its two 4-clauses:

$$\begin{aligned} f'(x_1,x_2,x_3,x_4,x_5,h,k) &= (h\vee x_1\vee\neg x_2)\wedge(\neg h\vee x_4\vee x_5)\wedge(x_2\vee\neg x_3\vee\neg x_4)\\ &\quad\wedge(k\vee\neg x_1\vee\neg x_2)\wedge(\neg k\vee x_3\vee\neg x_5)\wedge(\neg x_3\vee\neg x_4\vee x_5). \end{aligned}\tag{3.7}$$

Theorem 18 is interesting because it asserts that a polynomial-time algorithm for 3-SAT would be enough for the more general problem. With the addition of Theorem 17, we can conclude that all k-SAT languages, for $k\geq 3$, are equivalent to each other and to the more general SAT.

On the other hand, it can be shown that 2-SAT $\in\mathbf{P}$.

Simple reductions between graph languages

We already met CLIQUE; a strictly related problem is the one of finding an independent set in the graph:

INDEPENDENT SET (or simply INDSET) — Given an encoding of graph G and a number k, does G contain k nodes that are all *disconnected* from each other[8]?

The problem is almost the same, but we require the vertex subset to have *no* edges (while CLIQUE requires the subset to have *all possible* edges). Clearly, INDSET instances can be transformed into equivalent INDSET instances by simply complementing the edge set, which can be attained by negating the graph's adjacency matrix, which is clearly a polynomial time procedure in the graph's size (indeed, linear). Therefore, we can write both

$$\text{CLIQUE}\leq_p\text{INDSET}\qquad\text{and}\qquad\text{INDSET}\leq_p\text{CLIQUE}.$$

3.4.2 Example: reducing 3-SAT to INDSET

Let us see an example of reduction between two problems coming from different domains: boolean logic and graphs.

Theorem 19.

$$\textit{3-SAT}\leq_p\textit{INDSET}.$$

Proof. Let f be a 3-CNF formula. We need to transform it into a graph G and an integer k such that G has an independent set of size k if and only if f is satisfiable.

Let us represent each of the m clauses in f as a separate triangle (i.e., three connected vertices) of G, and let us label each vertex of the triangle as one of the clause's literals. Therefore, G contains $3m$ vertices organized in m triangles.

Next, connect every vertex labeled as a variable to all vertices labeled as the corresponding negated variable: every vertex labeled "$x1$" must be connected to every vertex labeled "$\neg x_1$" and so on. Fig. 3.1 shows the graph corresponding to the 3-CNF formula (3.7): each bold-edged triangle corresponds to one of the six clauses, with every node labeled with one of the literals. The dashed edges connect every literal with its negations.

It is easy to see that the original 3-CNF formula is satisfiable if and only if the graph contains an independent set of size $k=m$ (number of clauses). Given the structure of the graph, no more than one node per triangle can appear in the independent set (nodes in the same triangle are not independent), and if a literal appears in the independent set, then its negation does not (they would be connected by an edge, thus not independent). If the independent set has size m, then we are ensured that one literal per clause can be made true without contradictions. As an example, the six green nodes in Fig. 3.1 form an independent set and correspond to a truth assignment that satisfies f. $\square$

[8] `https://en.wikipedia.org/wiki/Independent_Set_(graph_theory)`

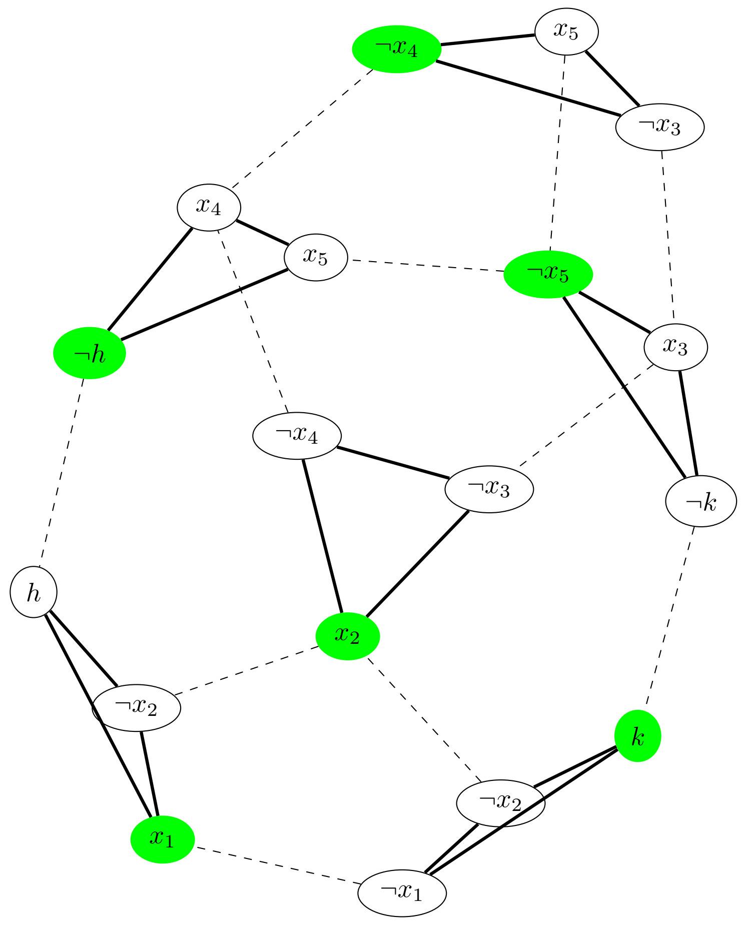


Figure 3.1: Reduction of the 3-CNF formula (3.7) to a graph for INDSET.

3.5 NP-hard and NP-complete languages

Definition 22. *A language L is said to be* **NP**-hard *if for every language $L' \in$ **NP** we have that $L' \leq_p L$.*

In this Section we will show that **NP**-hard languages exist, and are indeed fairly common. The definition just says that **NP**-hard languages are "harder" (in the polynomial reduction sense) than any language in **NP**: if we were able to solve any **NP**-hard language in polynomial time then, by this definition, we would have a polynomial solution to all languages in **NP**.

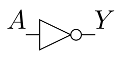
Furthermore, in this Section we shall see that the structure of **NP** is such that it is possible to identify a subset of languages that are "the hardest ones" within **NP**: we will call these languages ***NP**-complete*:

Definition 23. *A language $L \in$ **NP** that is **NP**-hard is said to be* **NP**-complete.

In particular, we will show that SAT is **NP**-complete.

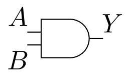
3.5.1 CNF and Boolean circuits

In order to prove the main objective of this part of the course, i.e. that SAT is NP-complete, we want to represent a computation of a NDTM as a CNF expression.



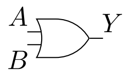
A	Y
1	0
0	1

$$\begin{aligned} & Y = \neg A \\ \equiv\ & (\neg Y \vee \neg A) \wedge (Y \vee A) \end{aligned}$$



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

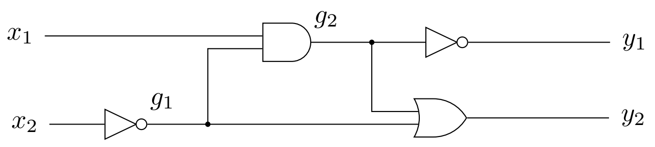
$$\begin{aligned} & Y = A \wedge B \\ \equiv\ & \big(\neg Y \vee (A \wedge B)\big) \wedge \big(Y \vee \neg(A \wedge B)\big) \\ \equiv\ & (\neg Y \vee A) \wedge (\neg Y \vee B) \wedge (Y \vee \neg A \vee \neg B) \end{aligned}$$



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

$$\begin{aligned} & Y = A \vee B \\ \equiv\ & \big(\neg Y \vee (A \vee B)\big) \wedge \big(Y \vee \neg(A \vee B)\big) \\ \equiv\ & (\neg Y \vee A \vee B) \wedge \big(Y \vee (\neg A \wedge \neg B)\big) \\ \equiv\ & (\neg Y \vee A \vee B) \wedge (Y \vee \neg A) \wedge (Y \vee \neg B) \end{aligned}$$

Figure 3.2: A NOT gate (top), an AND gate (middle) and an OR gate (bottom), their truth tables, and derivations of the CNF formulae that are satisfied if and only if their variables are in the correct relation (i.e., only by combinations of truth values shown in the corresponding table).



$$\begin{aligned} & (x_2 \vee g_1) \wedge (\neg x_2 \vee \neg g_1) \\ \wedge\ & (\neg g_2 \vee x_1) \wedge (\neg g_2 \vee g_1) \wedge (g_2 \vee \neg x_1 \vee \neg g_1) \\ \wedge\ & (y_1 \vee g_2) \wedge (\neg y_1 \vee \neg g_2) \\ \wedge\ & (\neg y_2 \vee g_1 \vee g_2) \wedge (y_2 \vee \neg g_1) \wedge (y_2 \vee \neg g_2) \end{aligned}$$

Figure 3.3: A Boolean circuit and its CNF representation: the CNF formula is satisfiable by precisely the combinations of truth values that are compatible with the logic gates.

A way to represent a Boolean formula as dependency of some outputs from some inputs is by means of a Boolean circuit, where logical connectives are replaced by gates. Fig. 3.2 shows the gates corresponding to the fundamental Boolean connectives, together with their truth tables and CNF formulae that are satisfiable by all truth assignments that are compatible with the gate.

We only consider *combinational* Boolean circuits, i.e., circuits that do not preserve states: there are no "feedback loops", and gates can be ordered so that every gate only receives inputs from previous gates in the order.

Any combinational Boolean circuit can be "translated" into a CNF formula, in the sense that the formula is satisfiable by all and only the combinations of truth values that satisfy the circuit. Given a Boolean circuit with n inputs $x_1, \dots, x_n$ and m outputs $y_1, \dots, y_m$ and l gates $G_1, \dots, G_l$:

- add one variable for every gate whose output is not an output of the whole circuit;
- once all gate inputs and outputs have been assigned a variable, write the conjunction of all CNF formulae related to all gates.

Fig. 3.3 shows an example: a Boolean circuit with 2 inputs, 2 outputs and 2 ancillary variables associated to intermediate gates, together with the corresponding CNF formula. This formula completely expresses the dependency between all variables in the circuit.

(to be continued)

Part II

Questions and exercises

Appendix A

Self-assessment questions

This chapter collects a few questions that students can try answering to assess their level of preparation.

A.1 Computability

A.1.1 Recursive and recursively enumerable sets

1. Why is every finite set recursive?
   (Hint: we need to check whether s is in a finite list)

2. Try to prove that if a set is recursive, then its complement is recursive too.
   (Hint: invert 0 and 1 in the decision function's answer)

3. Let S be a recursively enumerable set, and let algorithm $\mathcal{A}$ enumerate all elements in S. Prove that, if $\mathcal{A}$ lists the elements of S in increasing order, then S is recursive.
   (Hint: what if $n \notin S$? Is there a moment when we are sure that n will never be listed by $\mathcal{A}$?)

A.1.2 Turing machines

1. Why do we require a TM's alphabet Σ and state set Q to be finite, while we accept the tape to be infinite?

2. What is the minimum size of the alphabet to have a useful TM? What about the state set?

3. Try writing machines that perform simple computations or accept simply defined strings.

A.2 Computational complexity

A.2.1 Definitions

1. Why introduce non-deterministic Turing machines, if they are not practical computational models?

2. Why do we require reductions to carry out in polynomial time?

3. Am I familiar with Boolean logic and combinational Boolean circuits?

Appendix B

Exercises

Preliminary observations

Since the size of the alphabet, the number of tapes or the fact that they are infinite in one or both directions have no impact on the capabilities of the machine and can emulate each other, unless the exercise specifies some of these details, students are free to make their choices.

As for accepting or deciding a language, many conventions are possible. The machine may:

- erase the content of the tape and write a single "1" or "0";
- write "1" or "0" and then stop, without bothering to clear the tape, with the convention that acceptance is encoded in the last written symbol;
- have two halting states, `halt-yes` and `halt-no`;
- any other unambiguous convention;

with the only provision that the student writes it down in the exercise solution.

Exercise 1

For each of the following classes of Turing machines, decide whether the halting problem is computable or not. If it is, outline a procedure to compute it; if not, prove it (usually with with a reduction from the general halting problem). Unless otherwise stated, always assume that the non-blank portion of the tape is bounded, so that the input can always be finitely encoded if needed.
1.1) TMs with 2 symbols and at most 2 states (plus the halting state), starting from an empty (all-blank) tape.
1.2) TMs with at most 100 symbols and 1000000 states.
1.3) TMs that only move right;
1.4) TMs with a circular, 1000-cell tape.
1.5) TMs whose only tape is read-only (i.e., they always overwrite a symbol with the same one);

Hint — *Actually, only one of these cases is uncomputable...*

Solution 1

The following are minimal answers that would guarantee a good evaluation on the test.
1.1) The definition of the machine meet the requirements for the Busy Beaver game; Since we know the BB for up to 4 states, it means that every 2-state, 2-symbol machine has been analyzed on an empty tape, and its behavior is known. Therefore the HP is computable for this class of machines.
1.2) As we have seen in the lectures, 100 symbols and 1,000,000 states are much more than those needed to build a universal Turing machine $\mathcal{U}$. If this problem were decidable by a machine, say $\mathcal{H}_{1,000,000}$, then we could solve the general halting problem "does $\mathcal{M}$ halt on input s" by asking $\mathcal{H}_{1,000,000}$ whether $\mathcal{U}$ would halt on input (M,s) or not. In other words, we could reduce the general halting problem to it, therefore it is undecidable.
1.3) If the machine cannot visit the same cell twice, the symbol it writes won't have any effect on its future behavior. Let us simulate the machine; if it halts, then we output 1. Otherwise, sooner or later the machine will leave on its left all non-blank cells of the tape: from now on, it will only see blanks, therefore its behavior will only be determined by its state. Take into account all states entered after this moment; as soon as a state is entered for the second time, we are sure that the machine will run forever, because it is bound to repeat the same sequence of states over and over, and we can interrupt the simulation and output 0; if, on the other hand, the machine halts before repeating any state, we output 1.
1.4) As it has a finite alphabet and set of states (as we know from definition), the set of possible configurations of a TM with just 1000 cells is fully identified by (i) the current state, (ii) the current position, and (iii) the symbols on the tape, for a total of $|Q|\times 1000\times|\Sigma|^1000$ configurations. While this is an enormous number, a machine running indefinitely will eventually revisit the same configuration twice. So we just need to simulate a run of the machine: as soon as a configuration is revisited, we can stop simulating the machine and return 0. If, on the other hand, the simulation reaches the halt state, we can return 1.
1.5) Let $n=|Q|$ be the number of states of the machine. Let us number the cells with consecutive integer numbers, and consider the cells a and b that delimit the non-null portion of the tape. Let us simulate the machine. If the machine reaches cell $a-(n+1)$ or $b+n+1$, we will know that the machine must have entered some state twice while in the blank portion, therefore it will go on forever: we can stop the simulation and return 0. If, on the other hand, the machine always remains between cell $a-n$ and $b+n$, then it will either halt (then we return 1) or revisit some already visited configuration in terms of current cell and state; in such case we know that the machine won't stop because it will deterministically repeat the same steps over and over: we can then stop the simulation and return 0.

Exercise 2

2.1) Complete the proof of Theorem 9 by writing down, given a positive integer n, an n-state Turing machine on alphabet $\{0,1\}$ that starts on an empty (i.e., all-zero) tape, writes down n consecutive ones and halts below the rightmost one.
2.2) Test it for n=3.

Solution 2

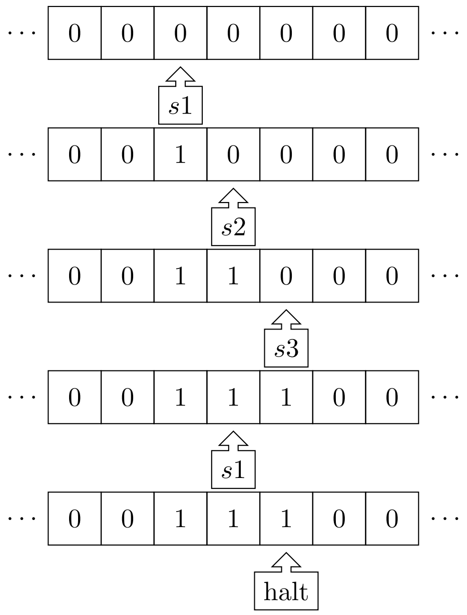
2.1) Here is a possible solution:

	0	1
s_1	1, right, s_2	1, right, halt
s_2	1, right, s_3	—
	$\vdots$	
s_i	1, right, s_{i+1}	—
	$\vdots$	
s_{n-1}	1, right, s_n	—
s_n	1, left, s_1	—

Entries marked by "—" are irrelevant, since they are never used. Any state can be used for the final move.
2.2) For $n = 3$, the machine is

	0	1
s_1	1, right, s_2	1, right, halt
s_2	1, right, s_3	—
s_3	1, left, s_1	—

Here is a simulation of the machine, starting on a blank (all-zero) tape:

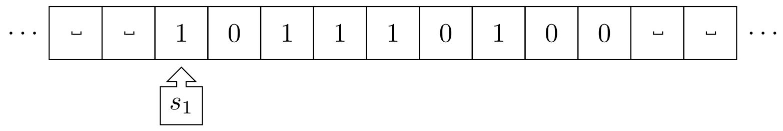


Exercise 3

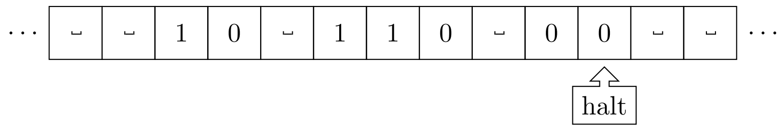
3.1) Write a Turing machine according to the following specifications:

- the alphabet is $\Sigma = \{␣, 0, 1\}$, where '␣' is the default symbol;
- it has a single, bidirectional and unbounded tape;
- the input string is a finite sequence of symbols in $\{0, 1\}$, surrounded by endless '␣' symbols on both sides;
- the initial position of the machine is on the leftmost symbol of the input string;
- every '1' that immediately follows '0' must be replaced with '␣' (i.e., every sequence '01' must become '0␣').
- the final position of the machine is at the righmost symbol of the output sequence.

For instance, in the following input case



the final configuration should be



You can assume that there is at least one non-'␣' symbol on the tape, but considering the more general case in which the input might be the empty string is a bonus.

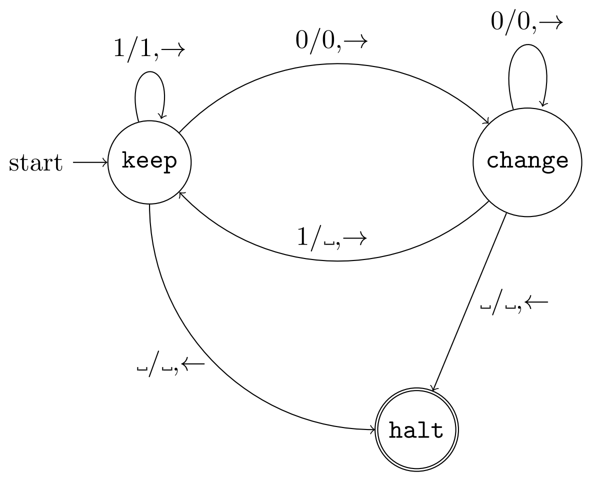
3.2) Show the sequence of steps that your machine performs on the input

"010011000111"

Solution 3

Two possible representations of the Turing machine are shown below; many other representations and transition rule sets are possible.

	␣	0	1
`keep`	␣/←/`halt`	0/→/`change`	1/→/`keep`
`change`	␣/←/`halt`	0/→/`change`	␣/→/`keep`


1/1,→
0/0,→
0/0,→
start
keep
change
1/␣,→
␣/␣,←
␣/␣,←
halt

$\boxed{\textbf{Exercise 4}}$

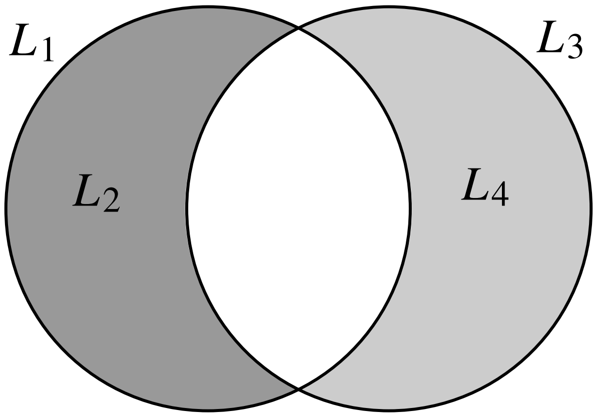
Let $\mathcal{M}$ represent a Turing Machine, let there be an encoding $s \to \mathcal{M}_s$ mapping string $s \in \Sigma^*$ to the TM $\mathcal{M}_s$ encoded by it. Finally, remember that in our notation $\mathcal{M}(x) = \infty$ means "$\mathcal{M}$ does not halt when executed on input x". Consider the following languages:

$$\begin{aligned}
L_1 &= \{s \in \Sigma^* \,|\, \exists x \mathcal{M}_s(x) \neq \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ halts on some inputs}\} \\
L_2 &= \{s \in \Sigma^* \,|\, \forall x \mathcal{M}_s(x) \neq \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ halts on all inputs}\} \\
L_3 &= \{s \in \Sigma^* \,|\, \exists x \mathcal{M}_s(x) = \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ doesn't halt on some inputs}\} \\
L_4 &= \{s \in \Sigma^* \,|\, \forall x \mathcal{M}_s(x) = \infty\} = \{s \in \Sigma^* \,|\, \mathcal{M}_s \text{ doesn't halt on any input}\}
\end{aligned}$$

4.1) Provide examples of TMs $\mathcal{M}_1, \dots, \mathcal{M}_4$ such that $\mathcal{M}_1 \in L_1, \dots, \mathcal{M}_4 \in L_4$.
4.2) Describe the set relationships between the four languages (i.e., which languages are subsets of others, which are disjoint, which have a non-empty intersection).

$\boxed{\textbf{Solution 4}}$

Observe that this exercise has very little to do with computability; however, being able to understand and answer it is a necessary prerequisite to the course. **4.1)** The machine that immediately halts ($s_0 = \texttt{HALT}$) is an example for L_1 and L_2. The machine that never halts (e.g., always moving right and staying in state s_0) is an example for L_3 and L_4.
4.2) If a machine always halts, it clearly halts on some inputs; therefore, $L_2 \subset L_1$ (equality is ruled out by the fact that there are machines that halt on some inputs and don't on others: $L_1 \cap L_3 \neq \emptyset$). With similar considerations, we can say that $L_4 \subset L_3$.
L_2 is disjoint from both L_3.
Also, observe that $L_2 = L_1 \setminus L_3$ and $L_4 = L_3 \setminus L_1$.
The relationship among the sets can be shown in the following diagram:



Exercise 5

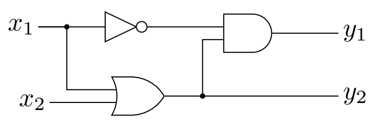
For each of the following properties of TMs, say whether it is semantic or not, and prove whether it is decidable or not.
5.1) $\mathcal{M}$ decides words with an 'a' in them.
5.2) $\mathcal{M}$ always halts within 100 steps.
5.3) $\mathcal{M}$ either halts within 100 steps or never halts.
5.4) $\mathcal{M}$ decides words from the 2018 edition of the Webster's English Dictionary.
5.5) $\mathcal{M}$ never halts in less than 100 steps.
5.6) $\mathcal{M}$ is a Turing machine.
5.7) $\mathcal{M}$ decides strings that encode a Turing machine (according to some predefined encoding scheme).
5.8) $\mathcal{M}$ is a TM with at most 100 states.

Solution 5

5.1) The property is semantic, since it does not depend on the specific machine but only on the language that it recognizes. The property is also non-trivial (it can be true for some machines, false for others), therefore it satisfies the hypotheses of Rice's theorem. We can safely conclude that it is uncomputable.
Note: the language "All words with an 'a' in them" is *computable. What we are talking about here is the "language" of all Turing machines that recognize it.*
5.2) Since we can always add useless states to a TM, given a machine M that satisfies the property, we can always modify it into a machine M' such that $L(M) = L(M')$, but that runs for more than 100 steps. Therefore the property is not semantic. It is also decidable: in order to halt within 100 steps, the machine will never visit more than 100 cells of the tape in either direction, therefore we "just" need to simulate it for at most 100 steps on all inputs of size at most 200 (a huge but finite number) and see whether it always halts within that term or not.
5.3) Again, the property is not semantic: different machines may recognize the same language but stop in a different number of steps. In this case, it is clearly undecidable: just add 100 useless states at the beginning of the execution and the property becomes "M never halts".
5.4) The property is semantic, since it only refers to the language recognized by the machine, and is clearly non-trivial. Therefore it satisfies Rice's Theorem hypotheses and is uncomputable. *Note: as in point 5.1, the language "all words in Webster's"* is *computable, but we aren't able to always decide whether a TM recognizes it or not.*
5.5) This is the complement of property 5.2, therefore not semantic and decidable.
5.6) The property is trivial, since all TMs trivially have it. Therefore, it is decidable by the TM that always says "yes" with no regard for the input.
5.7) The property is semantic because it refers to a specific language (strings encoding TMs). It is not trivial: even if the encoding allowed for all strings to be interpreted as a Turing machine, the only machines that possess the property would be those that recognize every string.
5.8) Deciding whether a machine has more or less than 100 states is clearly computable by just scanning the machine's definition and counting the number of different states. The property is not semantic.

Exercise 6

Consider the following Boolean circuit:



6.1) Write down the CNF formula that is satisfied by all and only combinations of input and output values compatible with the circuit.
6.2) Is it possible to assign input values to x_1, x_2 such that $y_1 = 0$ and $y_2 = 1$? Provide a CNF formula that is satisfiable if and only if the answer is yes.
6.3) Is it possible to assign input values to x_1, x_2 such that $y_1 = 1$ and $y_2 = 0$? Provide a CNF formula that is satisfiable if and only if the answer is yes.

Solution 6

6.1) Let g_1 be the variable associated to the NOT gate; the other two gates are already associated to the circuit's outputs. The formula, obtained by combining the equations in Fig. 3.2 is therefore:

$$\begin{aligned}
f(x_1,x_2,g_1,y_1,y_2) &= (\neg y_2 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee y_2) \wedge (\neg x_2 \vee y_2)\\
&\quad \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g_1)\\
&\quad \wedge (\neg y_2 \vee \neg g_1 \vee y_1) \wedge (\neg y_1 \vee y_2) \wedge (\neg y_1 \vee g_1).
\end{aligned}$$

The first line describes the OR gate, the second the NOT, the thirs the AND.
6.2) Let us set $y_1 = 0$ and $y_2 = 1$ in f and simplify:

$$\begin{aligned}
f'(x_1,x_2,g_1) &= f(x_1,x_2,g_1,0,1)\\
&= (\emptyset \vee x_1 \vee x_2) \wedge \cancel{(\neg x_1 \vee 1)} \wedge \cancel{(\neg x_2 \vee 1)}\\
&\quad \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g1)\\
&\quad \wedge (\emptyset \vee \neg g_1 \vee \emptyset) \wedge \cancel{(1 \vee 1)} \wedge \cancel{(1 \vee g_1)}\\
&= (x_1 \vee x_2) \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g1) \wedge \neg g_1.
\end{aligned}$$

Note that f' is satisfiable: the last clause obviously requires $g_1 = 0$, after which the second clause implies $x_1 = 1$ and the value of x_2 becomes irrelevant. Therefore, by just setting $x_1 = 1$ the circuit will provide the required output.
6.3) Let us perform the substitution:

$$\begin{aligned}
f''(x_1,x_2,g_1) &= f(x_1,x_2,g_1,0,1)\\
&= \cancel{(1 \vee x_1 \vee x_2)} \wedge (\neg x_1 \vee \emptyset) \wedge (\neg x_2 \vee \emptyset)\\
&\quad \wedge (x_1 \vee g_1) \wedge (\neg x_1 \vee \neg g_1)\\
&\quad \wedge \cancel{(1 \vee \neg g_1 \vee 1)} \wedge \overbrace{(0 \vee 0)}^{\text{unsatisfiable}} \wedge (\emptyset \vee g_1),
\end{aligned}$$

which, because of the second-to-last clause, cannot be satisfied. Therefore, the circuit cannot have the required output.