# Computability and Computational Complexity
Academic year 2026–2027, first semester
Lecture notes

Mauro Brunato

Version: 2026-11-16

**Caveat Lector**

These *very schematic and possibly incomplete* lecture notes have been drafted during the past editions of the course; they willl be updated during the 2026-2027 edition to keep track of the lectures.
Their main purpose is to keep track of what has been said during the lectures.
Many topics in these notes might be explained as they were presented in past editions, in a slightly different way with respect to this year's lectures. **This is a feature, not a bug**: having the same topic illustrated twice with different wordings might help clarify them.
Let it be clear that **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 also contains 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 to the one reported on the webpage.

# TODO and Changelog

This log (in reverse temporal order, most recent change first) highlights the differences between the current edition of the notes and the previous ones.

**2026-11-16**

- Today's lesson marked on the text.

**2026-11-11**

- Text updated and marked according to yesterday's lesson.
- Exercises 25–35 have been brought to front and are now feasible.

**2026-11-09**

- Text updated and marked according to today's lesson.

**2026-11-04**

- Text updated and marked according to yesterday's lesson; moved Section 3.3.2 to its new position.

**2026-11-02**

- Text updated and marked according to today's lesson.

**2026-10-29**

- Text updated and marked according to the 2026-10-27 lesson.

**2026-10-26**

- Added markers for today's lesson.
- Published solutions to exercises up to 24.

**2026-10-20**

- Added property $\mathcal{P}_2$ to exercise 11 to illustrate semantic, non-trivial properties that are RE.
- Added markers for today's lesson.
- Modified the notation of Section 2.1 to align it to previous chapters.
- Added/moved some unimportant (but useful for self-assessment) considerations to the "Fun facts" Section 2.2.

**2026-10-19**

- Added exercise 22 proposed at the end of today's lecture.

**2026-10-13**

- Exercises 7–21 moved up (but their solutions aren't shown yet). Now all exercises up to 21 should be feasible with the current knowledge.
- Text updated and marked according to today's lecture.

**2026-10-12**

- Text updated according to today's lecture.

**2026-10-06**

- Exercise 6 appeared far down the list, but it is feasible; moved up.
- Published solutions to exercises 1–6.
- Simplified notation for exercise 4.
- Adjusted and marked the theory part with today's topics.

**2026-10-05-b (after the lecture)**

- Text updated according to today's lecture.

**2026-10-05 (before the lecture)**

- Feasible exercises moved to the top of the list (1–5).

**2026-10-02**

- Notes reorganized and notation adjusted according to the last two lessons (2026-09-28/29).

**2026-09-22**

- Proof of Theorem 3 reviewed.

**2026-09-21**

- Added Fig. 1.1 as visual support to the proof of Theorem 2.
- Added Section 1.2 to summarize the ideas up to this moment and to clarify definitions.
- Moved Section about Collatz sequences to the "additional" section, to be retrieved when appropriate.
- Exercise reordering: exercise 1 is now in its final place.

**2026-09-16**

- Adapted initial part to current year's notation.
- Added markers for beginning and end of lecture.
- Added index of lectures with page references.

## 2026-09-06

- Collected exercises from last year exams.
- Removed solutions to exercises; they will be added after class discussion.

## 2026-09-03

- Initial version, composed of the final draft frome the 2025-26 course.

# Contents


**I Lecture notes** **6**

**1 Computability** **7**
1.1 Basic definitions and examples . . . 7
1.1.1 Non-computable functions . . . 8
1.2 Summary and definitions . . . 10
1.3 A computational model: the Turing machine . . . 11
1.3.1 Examples . . . 12
1.3.2 Computational power of the Turing Machine . . . 12
1.3.3 Universal Turing machines . . . 13
1.4 Computable and non-computable functions . . . 14
1.4.1 The Church-Turing thesis . . . 14
1.4.2 Turing machines are countable . . . 14
1.4.3 A few formal (re)definitions . . . 14
1.4.4 Finding a non-computable function . . . 16
1.4.5 Reductions . . . 17
1.4.6 Recursively enumerable decision functions and languages . . . 18
1.4.7 Another uncomputable function: the Busy Beaver game . . . 22
1.4.8 ...So what? . . . 23
1.5 Rice's Theorem . . . 24

**2 More undecidable problems** **26**
2.1 Kolmogorov complexity . . . 26
2.1.1 Dependence on the underlying computational model . . . 27
2.1.2 Non-computability of Kolmogorov complexity . . . 27
2.2 "Fun facts" about computability . . . 28
2.2.1 Provably undecidable machines . . . 28
2.2.2 More consequences of the Halting Problem non-computability . . . 29
2.2.3 Kolmogorov complexity, the halting problem and paradoxes . . . 30
2.2.4 Recursive enumerability of properties . . . 30

**3 Complexity classes: P and NP** **31**
3.1 Definitions . . . 31
3.2 Time classes . . . 32
3.2.1 Polynomial-time and exponential-time languages . . . 33
3.2.2 **P** is good and **EXP** is bad . . . 34
3.2.3 Boolean formulas and the conjunctive normal form . . . 35
3.3 **NP** languages . . . 36
3.3.1 Non-deterministic Turing Machines . . . 37
3.3.2 An asymmetry in the definition of **NP**: the class **coNP** . . . 38
3.4 Reductions and hardness . . . 39

3.4.1 Simple examples . . . 40
3.4.2 Example: reducing 3-SAT to INDSET . . . 41
3.4.3 Another example: reducing SAT to an arithmetic problem . . . 41
3.5 **NP**-hard and **NP**-complete languages . . . 43
3.5.1 Boolean circuits and 3-CNF formulas: the Tseytin transformation . . . 43
3.5.2 Using Boolean circuits to express Turing Machine computations . . . 46
3.6 More **NP**-complete languages . . . 51
3.7 Arithmetic problems . . . 53
3.7.1 SUBSET SUM . . . 53
3.7.2 KNAPSACK . . . 55
3.8 Problems on graphs: paths and traveling salesmen . . . 55
3.8.1 Hamiltonian paths . . . 56
3.8.2 Directed Hamiltonian cycles . . . 59
3.8.3 Undirected Hamiltonian cycles . . . 59
3.8.4 The Traveling Salesman Problem . . . 61
3.8.5 Relationship between **P**, **NP** and **coNP** . . . 61

**4 Other complexity classes** **63**
4.1 The exponential time classes . . . 63
4.1.1 The "Restricted" Halting Problem . . . 63
4.2 Space complexity classes . . . 65
4.2.1 Logarithmic space classes: **L** and **NL** . . . 65
4.2.2 **NL**-completeness of STCON . . . 67
4.2.3 Polynomial space: **PSPACE** and **NPSPACE** . . . 68
4.2.4 The Quasi-polynomial time class . . . 70

**II Additional material (not required)** **71**

**5 Topics from previous editions** **72**
5.1 A set which we are (currently) unable to enumerate . . . 72
5.1.1 Collatz sequences . . . 72
5.1.2 Decidability of the Collatz property . . . 72
5.2 Post Correspondence Problem . . . 73
5.2.1 Undecidability of the Modified PCP . . . 75
5.2.2 Undecidability of the Post Correspondence Problem . . . 79
5.3 Enhancing and restricting TMs . . . 79
5.3.1 Oblivious Turing Machines . . . 79
5.3.2 Allowing infinite states . . . 81
5.4 A relationship between exponential and polynomial time classes . . . 81
5.5 The Merkle-Hellman cryptosystem . . . 82
5.5.1 $k$-VERTEX COLORING for $k > 3$ . . . 83
5.6 Randomized complexity classes . . . 84
5.6.1 The classes **RP** and **coRP** . . . 84
5.6.2 Zero error probability: the class **ZPP** . . . 86
5.6.3 Symmetric probability bounds: classes **BPP** and **PP** . . . 88
5.7 Quantum computing . . . 89
5.8 Function problems . . . 92
5.8.1 Relationship between functional and decision problems . . . 93
5.9 Interactive proof systems . . . 93
5.9.1 An example: GRAPH ISOMORPHISM . . . 93
5.9.2 The Arthur-Merlin protocol . . . 94

5.9.3 The Interactive Polynomial protocol class . . . 94
5.10 Zero-knowledge proofs . . . 95

**6 Further directions** **97**
6.1 About **NP** . . . 97
6.2 Above **NP** . . . 97
6.3 Other computational models . . . 97

## III Questions and exercises 98

**A Self-assessment questions** **99**
A.1 Computability . . . 99
A.1.1 Recursive and recursively enumerable sets . . . 99
A.1.2 Turing machines . . . 99
A.1.3 Rice's Theorem . . . 99
A.2 Computational complexity . . . 99
A.2.1 Definitions . . . 99
A.2.2 **P** vs. **NP** . . . 100
A.2.3 Other complexity classes . . . 100
A.2.4 General discussion . . . 100

**B Exercises** **101**

**C Old exercises** **148**

**List of lectures** **150**

**Index** **151**

# Part I

# Lecture notes

# Chapter 1

# Computability

## 1.1 Basic definitions and examples

2026-09-15 start

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 $\Sigma$ denote a finite set of *symbols*. $\Sigma$ 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:

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

As a shorthand, instead of representing tuples of symbols in the formal notation $(s_1, s_2, \ldots, s_n)$ 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 $\Sigma$:

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

It is worthwhile to note that $\Sigma^*$, while being infinite in itself, only contains *finite* sequences of symbols. Moreover, for every $n \in \mathbb{N}$, $\Sigma^n$ 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 co-domain 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 are not.

**Decision functions and sets**

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

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

(technically, the set is the "inverse image" of 1 throught the function $f$: $S_f = f^{-1}(1)$). Conversely, given a string subset $S \subseteq \Sigma^*$, we can always define the function that decides over elements of the set, called the "characteristic function" of $S$:

$$\begin{aligned} \chi_S : \Sigma^* &\to \{0, 1\} \\ w &\mapsto \begin{cases} 1 & \text{if } w \in S \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

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 co-finite[2] set, and any function that decides on them;
- any function studied in a basic Algorithms course (sorting, hashing, spanning trees on graphs...).

We will see that the set of "computer programs" is too small with respect to the set of all decision functions; therefore, for some functions there is no program able to compute them. In the first part of this course we will first define what is a "computer program" and we will proceed to identify a non-computable decision function.

### 1.1.1 Non-computable functions

The existence of decision functions that cannot be computed is quite easy to prove: we can show that the set of possible computer programs, albeit infinite, is too "small" (in a sense that will be made rigorous in a few moments) to contain a program for every possible decision function.

As will often be the case in the future, let us decide that $\Sigma$ is any alphabet we are comfortable with to describe all objects at hand; e.g., we will consider all characters representable by Unicode.

First of all, let us notice that we can write a program that prints out every possible string in $\Sigma^*$. A set with this property is called *countable* because by writing this list we associate a natural number to each element, i.e. a one-to-one map from $\mathbb{N}$ to our set (in this case, $\Sigma^*$).

**Theorem 1.** *$\Sigma^*$ is countable.*

[1] `https://en.wikipedia.org/wiki/Recursive_set`

[2] A set is *co-finite* when its complement is finite.

| Function enumerated by string index | Function argument: $\varepsilon$ | a | b | c | aa | ab | ac | ba | bb | bc | ca | cb | cc | aaa | aab | aac | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f_\varepsilon$ | (1) | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | ... |
| $f_{\mathtt{a}}$ | 0 | (1) | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | ... |
| $f_{\mathtt{b}}$ | 1 | 0 | (0) | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | ... |
| $f_{\mathtt{c}}$ | 0 | 0 | 1 | (1) | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | ... |
| $f_{\mathtt{aa}}$ | 1 | 1 | 0 | 0 | (0) | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | ... |
| $f_{\mathtt{ab}}$ | 0 | 1 | 0 | 1 | 0 | (1) | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | ... |
| $f_{\mathtt{ac}}$ | 1 | 0 | 0 | 1 | 0 | 0 | (1) | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | ... |
| $f_{\mathtt{ba}}$ | 1 | 0 | 0 | 0 | 0 | 0 | 1 | (1) | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | ... |
| $f_{\mathtt{bb}}$ | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | (0) | 1 | 0 | 0 | 0 | 1 | 0 | 0 | ... |
| $f_{\mathtt{bc}}$ | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | (1) | 1 | 0 | 0 | 0 | 1 | 1 | ... |
| $f_{\mathtt{ca}}$ | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | (0) | 1 | 1 | 1 | 1 | 0 | ... |
| $f_{\mathtt{cb}}$ | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | (0) | 0 | 0 | 0 | 1 | ... |
| $f_{\mathtt{cc}}$ | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | (1) | 0 | 1 | 0 | ... |
| $f_{\mathtt{aaa}}$ | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | (1) | 1 | 0 | ... |
| $f_{\mathtt{aab}}$ | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | (1) | 1 | ... |
| $f_{\mathtt{aac}}$ | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | (0) | ... |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |
| $h$ | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | ... |

Figure 1.1: Visual proof of Theorem 2: suppose that we have a complete enumeration of decision functions, indexed with strings: the infinite table above shows a hypothetical example reporting every value for every function for alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. We can always define a new decision function $h$ such that, for any string $s \in \Sigma^*$, $h(s) \neq f_s(s)$ (note that the values of $f_s(s)$ are the circled ones, on the diagonal); for example, $h(\mathtt{aa}) = 1 \neq 0 = f_{\mathtt{aa}}(\mathtt{aa})$. Therefore, $h$ is different from all functions in the enumeration, hence the enumeration is not complete.

*Proof.* An enumerating procedure works as follows: first we count the only string in $\Sigma^0$, then the strings in $\Sigma^1$, then those in $\Sigma^2$ (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., $\Sigma^*$ is countable. □

But computer programs can be represented as strings, and it is easy to see that we can restrict the above defined procedure to only output a string if it's a valid program (in some pre-agreed programming language), therefore

**Corollary 1.** *The set $P \subseteq \Sigma^*$ of all computer programs is countable.*

It is easy to understand that, even if we restrict our interest to decision functions, "almost all" functions cannot be computed by a computer program. In fact, as the following theorem 2 and the previous corollary 1 show, there are simply too many functions to be able to define an algorithm for each of them.

**Theorem 2.** *The set of decision functions $\left\{f : \Sigma^* \to \{0,1\}\right\}$ is uncountable.*

*Proof.* First of all, observe that the result of Theorem 1 tells us that we can use strings instead of numbers as indexes of a set, because they are in a one-to-one correspondence.

Our theorem states an impossibility, as will often be the case in this course: that we cannot enumerate all decision functions. This kind of theorems can be proved by contradiction: we start by assuming that the enumeration *is* possible, and we will show that this leads to some contradiction. You can refer to Fig. 1.1 for visual support.

So, by contradiction, let's assume that a complete mapping exists from the $\Sigma^*$ to the set of decision functions; i.e., there is a mapping $s \mapsto f_s$ that enumerates all functions. This means that, no matter which decision function $g : \Sigma^* \to \{0, 1\}$, we will always find an index $s \in \Sigma^*$ such that the "$s$-th" object in our enumeration $f_s$ is precisely $g$.

Now we proceed to define a decision function that cannot appear in the enumeration. Let

$$\begin{aligned} h : \Sigma^* &\to \{0, 1\} \\ s &\mapsto 1 - f_s(s). \end{aligned}$$

By its own definition, for every index $s \in \Sigma^*$, function $h$ differs from $f_s$ on the value it is assigned for $s$ (if $f_s(s) = 0$ then $h(s) = 1 - f_s(s) = 1 - 0 = 0$, and vice versa). Therefore, contrary to the assumption, the enumeration could not be complete because it excluded at least function $\hat{f}$. □

Lemma 2 is an example of *diagonal argument*[3], introduced by Cantor in order to prove the uncountability of real numbers: it focuses on the "diagonal" values (in our case $f_s(s)$, by using the same string as function index and as argument), and it makes a new object that systematically differs from all that are listed.

Therefore, whatever way we choose to enumerate algorithms and to associate them with decision functions, we will inevitably leave out some functions. Hence,

**Corollary 2.** *There are uncomputable decision functions.*

In fact, we can say that "almost all" functions are uncomputable, in the sense that only a small (although infinite), countable subset of them can be associated to an algorithm.

Note that, for the moment being, these assertions are not as precise as we would like them to be: we rely on the assumption that algorithms are expressed as strings of symbols, but we haven't stated exactly what an algorithm is, yet. To this end, later we will introduce our computational model, Turing Machines.

2026-09-15
end

## 1.2 Summary and definitions

2026-09-21
start

Let's put our facts in order and add some definitions:

- An *alphabet* $\Sigma$ is any finite set; its elements $\sigma \in \Sigma$ are called *symbols*.
- Strings are tuples of symbols; their set is called $\Sigma^*$.
- There is a one-to-one correspondence between $\mathbb{N}$ and $\Sigma^*$ (i.e., an *enumeration* of strings). I.e., $\Sigma^*$ is a *countable* set.
- A subset of $\Sigma^*$ (i.e., a set of strings) is called a *language*; it can be either finite or infinite.
- The power set of $\Sigma^*$ ($\mathcal{P}(\Sigma^*)$, the set of all subsets of $\Sigma^*$) is thus the set of all languages.
- There is a canonical, one-to-one correspondence between $\mathcal{P}(\Sigma^*)$ and the set of decision functions on strings.
- The set of decision functions on strings, and hence $\mathcal{P}(\Sigma^*)$, is uncountable (any enumeration of its elements leaves out some).
- A language $L$ is *computable* if there is a program in some language that, given an arbitrary string $s \in \Sigma^*$ as input tells us whether $s \in L$ or not (this definition requires some clarification as to what is an acceptable programming language: we will came to this in the next Section).

[3] `https://en.wikipedia.org/wiki/Cantor%27s_diagonal_argument`

- Since programs are strings, they form a countable set; however, languages are uncountably many; therefore, some languages are not computable.
- Likewise, a language $L$ can be said to be *definable* if there is a logical predicate $\mathcal{P}$ (a proposition written in mathematical and logical terms) such that $L = \{s \in \Sigma^* | \mathcal{P}(s) \text{ is true}\}$. For instance,

  $$\text{PRIME NUMBERS} = \{s \in \Sigma^* | \text{s represents natural number } n \wedge \forall m \in \mathbb{N}\ 1 < m < n \Rightarrow m \not| n\}.$$

- Again, predicates are just strings of mathematical symbols, hence countable. Therefore, there are many languages that are not definable.

The purpose of the following sections is to formalize the notion of computability given above, by introducing a very simple but universal computational framework, and identify a language (or, equivalently, a decision function) that is precisely defined, but still not computable.

## 1.3 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 one which is more similar to our intuitive notion. A Turing Machine[4] is defined by:

- a finite alphabet $\Sigma$, 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:

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

  This set of rules is also called *transition function*.

Initially, all cells contain the default symbol, with the exception of a finite number; the non-blank portion of the tape represents 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 $(q, \sigma)$:

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

The machine writes the symbol $\sigma'$ 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 $n$.

2026-09-21
end

### 1.3.1 Examples

2026-09-22
start

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.

### 1.3.2 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 3** ($k$-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 $O\big(T(n)^2\big)$.*

*Proof.* See Arora-Barak, Claim 1.9 in the public draft, for a better explanation and an illustration[5].

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

[5] Be aware that the Arora-Barak definition of a TM is slightly different from ours: their default model assumes a tape that infinitely extends in just one direction, and the existence of separate input (read-only) and output (write-only) tapes. These details don't change the overall picture, though.

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 and perform 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 $kT(n) = O\big(T(n)\big)$ steps. Therefore, the simulation of a single step of $\mathcal{M}$ will take at most $2kT(n)$ steps by $\mathcal{M}'$, therefore the whole simulation takes $O\big(T(n)^2\big)$ steps. □

2026-09-22 end

2026-09-28 start

**Size of the alphabet**

The number of symbols that can be written on a tape (the size of the alphabet $\Sigma$) 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 alphabet is enough to simulate any TM:

**Theorem 4** (Emulation by a two-symbol Turing Machine)**.** *If a Turing machine $\mathcal{M}$ with a $k$-symbol alphabet $\Sigma$ 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 $\Sigma$ can be encoded by $\lceil \log_2 k \rceil$ binary digits from $\Sigma'$. 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 jump to memory" and so on can be emulated.

### 1.3.3 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."

## 1.4 Computable and non-computable functions

### 1.4.1 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 result about TMs to every physical computational device.

### 1.4.2 Turing machines are countable

As seen in Section 1.3.3, a universal Turing machine $\mathcal{U}$ works on a string representation of the Turing machine that it needs to emulate. Therefore, if TM is the set of all Turing machines (on a given alphabet) we have a mapping

$$\begin{aligned} \hat{\cdot} : \mathrm{TM} &\to \Sigma^* \\ \mathcal{M} &\mapsto \hat{\mathcal{M}}, \end{aligned}$$

i.e., $\hat{\mathcal{M}}$ is the string representing the machine $\mathcal{M}$ in the chosen notation.

Observe that $\hat{\cdot}$ is a total function (every TM has a string description), but it is not invertible, because many strings are not descriptions of TMs (they don't have the proper syntax). In other words, its inverse

$$\begin{aligned} \lfloor \cdot \rfloor : \Sigma^* &\to \mathrm{TM} \\ m &\mapsto \lfloor m \rfloor, \end{aligned}$$

where $\lfloor m \rfloor$ is the Turing machine represented by string $m$, is not a total function. However, we can make it total by assuming that all "illegal" strings represent the same arbitrary machine, e.g. the one that immediately halts without changing the tape content. With this assumption, every string corresponds to a Turing machine, and the map $\lfloor \cdot \rfloor$ becomes surjective.

The existence of the two functions $\hat{\cdot}$ and $\lfloor \cdot \rfloor$ allows us to speak interchangeably of strings and Turing machines and implicitly tells us that the set of Turing machines, although infinite, is countable.

2026-09-28 end

### 1.4.3 A few formal (re)definitions

2026-09-29 start

At this point, the results from Section 1.1.1 can be given a precise and formal meaning by just replacing the generic words "algorithm" or "computer program" with the more precise term "Turing Machine": since every TM can be encoded into a string, TMs can only form a countable set, too few to be able to express all decision functions.

**Definition 1** (input of a TM)**.** *Given a Turing machine $\mathcal{M}$ its* input *is the non-default portion of its tape before execution; unless otherwise stated, we assume that the initial position on the tape lies on some convenient symbol within the input.*

Let $x \in \Sigma^*$ be the input of $\mathcal{M}$; then we can see the execution of $\mathcal{M}$ as a function that maps the input to whatever will be the outcome of the computation.

**Definition 2** (output of a TM)**.** *Given a string $x$ and a Turing machine $\mathcal{M}$, the outcome of $\mathcal{M}$'s computation on input $x$ is written $\mathcal{M}(x)$. If the computation halts, then its output can be defined as one of the following:*

- *the string contained in the non-default portion of the tape, or*
- *in the case of multiple halting states, the specific state reached by the $\mathcal{M}$.*

*If the computation doesn't halt, then the outcome is represented by the infinity symbol, $\infty$.*

For instance, a machine that computes a decision function might halt after leaving the string "`yes`" or "`no`" on the tape (or just "`1`" or "`0`"); or it might end in either one of two distinct halting states, "$\text{HALT}_{\text{yes}}$" or "$\text{HALT}_{\text{no}}$".

Observe that the introduction of $\infty$ as a (purely formal) outcome of the computation allows us to treat every Turing machine as a total function

$$\begin{array}{rcl} \mathcal{M} : \Sigma^* & \to & \Sigma^* \cup \{\infty\} \\ s & \mapsto & \mathcal{M}(s) \end{array}$$

Given these definitions, given a decision function $f$ on strings and a Turing machine $\mathcal{M}$ we are allowed to write things like

$$\mathcal{M}(s) = f(s)$$

if the Turing machine halts on input $s$ and its output agrees with the function $f$.

We are now ready to rewrite in a formally sound fashion the definition:

**Definition 3** (computing and deciding)**.** *Let $f : \Sigma^* \to \Sigma^*$ a function on strings. We say that the Turing machine $\mathcal{M}$* computes *it if*

$$\forall s \in \Sigma^* \quad \mathcal{M}(s) = f(s),$$

*i.e., the machine $\mathcal{M}$ always halts and provides the correct output.*
*If $f$ is a decision function (i.e., its codomain is binary), then we may also say that $\mathcal{M}$* decides *$f$.*
*Finally, if we have a language $L \subset \Sigma^*$ then we say that the machine $\mathcal{M}$* computes *(or* decides*) it if it computes its characteristic function $\chi_L$, i.e., if*

$$\forall s \in \Sigma^* \quad \mathcal{M}(s) = \chi_L(s) = \begin{cases} 1 & \text{if } s \in L \\ 0 & \text{otherwise.} \end{cases}$$

Finally, we have a sound definition of what a computable function is:

**Definition 4.** *A function $f : \Sigma^* \to \Sigma^*$ (and similarly a decision function $f : \Sigma^* \to \{0,1\}$ or a language $L \subseteq \Sigma^*$) is* computable *or* recursive *if there is a Turing machine that computes it (in the sense defined above).*

A computable decision function or a computable language may also be called *decidable*[6].

[6] In other words, the terms *computable*, *recursive* and *decidable* are synonyms, although the latter is reserved to binary outcomes.

### 1.4.4 Finding a non-computable function

Let us now set out to find a function that cannot be computed by any TM (and therefore, by the Church-Turing thesis, by any feasible computational model).

With the above definitions in mind, we can design a decision function whose outcome differs from that of any TM. We employ a diagonal technique akin to the proof of Lemma 2: for any string $\alpha \in \Sigma^*$, we define our function to differ from the output of the TM encoded by $\alpha$ on input $\alpha$ itself.

**Theorem 5.** *Given an alphabet $\Sigma$ and an encoding $\alpha \mapsto \lfloor\alpha\rfloor$ of TMs in that alphabet, the function*

$$UC(\alpha) = \begin{cases} 0 & \text{if } \lfloor\alpha\rfloor(\alpha) = 1 \\ 1 & \text{otherwise} \end{cases} \qquad \forall \alpha \in \Sigma^*$$

*is not computable.*

*Proof.* Let $\mathcal{M}$ be any TM, and let $m = \hat{\mathcal{M}} \in \Sigma^*$ be its encoding. 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 $\lfloor\alpha\rfloor$ on input $\alpha$ 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 6** (Halting problem)**.** *Given an alphabet $\Sigma$ and a encoding $\alpha \mapsto \lfloor\alpha\rfloor$ of TMs in that alphabet, the function*

$$HALT(s,t) = \begin{cases} 0 & \text{if } \lfloor s\rfloor(t) = \infty \\ 1 & \text{otherwise} \end{cases} \forall (s,t) \in \Sigma^* \times \Sigma^* (= \Sigma^*),$$

*which returns 1 if and only if machine $\lfloor s\rfloor$ halts on input $t$, is not computable.*

*Proof.* Let's proceed by contradiction. Suppose that we have a machine $\mathcal{H}$ which computes $\text{HALT}(s,t)$ (i.e., when run on a tape containing a string $s$ encoding a TM and a string $t$, always halts returning 1 if machine $\lfloor s\rfloor$ 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 $\alpha$ 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 $\lfloor\alpha\rfloor(\alpha)$ would terminate or not. Two outcomes are possible:
  - If the output of $\mathcal{H}$ is zero, then we know that the computation of $\lfloor\alpha\rfloor(\alpha)$ 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 $\lfloor\alpha\rfloor(\alpha)$ 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 $\alpha$ 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 Halting 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:

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

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

**Theorem 7.** *$HALT_\varepsilon$ is not computable.*

*Proof.* By contradiction, suppose that there is a machine $\mathcal{H}_\varepsilon$ 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 $\text{HALT}(s, t)$ for a non-empty input string $t$. We can transform the computation $\lfloor s \rfloor(t)$ on a computation
$lfloors' \rfloor(\varepsilon)$ on an empty tape where $s'$ contains the whole encoding $s$, but preceded by 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 $\text{HALT}(s, t)$, which would therefore be computable. □

Again, the result was obtained by "reducing" a known non-computable problem, HALT, to the newly introduced one, $\text{HALT}_\varepsilon$.

### 1.4.5 Reductions

Note that two results in the past sections (Theorems 6 and 7 made use of similar arguments: "If $A$ were computable, then we could use it to solve $B$; however, we know that $B$ is not computable, therefore $A$ isn't too." Now we want to formalize such reasoning scheme.

**Definition 5.** *Let $f_1 : \Sigma^* \to \Sigma^*$ and $f_2 : \Sigma^* \to \Sigma^*$ be two (possibly decision) functions. A function*

$$g : \Sigma^* \to \Sigma^*$$

*is said to be a* reduction *from $f_1$ to $f_2$ if*

$$\forall s \in \Sigma^* \quad f_1(s) = f_2(g(s)).$$

*Similarly, let $L_1 \subset \Sigma^*$ and $L_2 \subset \Sigma^*$ be two languages. A function*

$$g : \Sigma^* \to \Sigma^*$$

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

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

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

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

**Definition 6.** *We say that $g : \Sigma^* \to \Sigma^*$ is a* Turing reduction *from $f_1 : \Sigma^* \to \Sigma^*$ to $f_2 : \Sigma^* \to \Sigma^*$ if it is a reduction from $f_1$ to $f_2$ and it is computable.*
*Similarly, we say that $g : \Sigma^* \to \Sigma^*$ is a* Turing reduction *from $L_1 \subset \Sigma^*$ to $L_2 \subset \Sigma^*$ 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 $L_1 <_T L_2$. 2026-09-29

Note that we do *not* require $g$ to have any specific property such as being injective or surjective: end
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 $L_2$. 2026-10-05

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

**Theorem 8.** *Let languages $L_1$ and $L_2$ and function $g$ be such that $L_1 <_f L_2$; then*

1. *if $L_2$ is decidable and $g$ is computable, then $L_1$ is decidable too;*
2. *if $L_1$ is non-decidable and $g$ is computable, then $L_2$ is non-decidable too;*
3. *if $L_1$ is non-decidable and $L_2$ is decidable, then $g$ is non-computable.*

*Proof.* The first point is proven by showing that, if we have a machine for $g$ 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 $\mathcal{M}_g$ and an execution of $\mathcal{M}_{L_2}$, i.e. computes $\mathcal{M}(s) = \mathcal{M}_{L_2}\big(\mathcal{M}_g(s)\big)$, decides $L_1$ by definition of $f$.

The other two points follow by contradiction.

□

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

Observe that the proofs of Theorems 6 and 7 follow the second scheme of Theorem 8.

### 1.4.6 Recursively enumerable decision functions and languages

Let us define the set R of all recursive languages:

$$R = \{L \subseteq \Sigma^* : \exists \mathcal{M} \in TM \text{ s.t. } \mathcal{M} \text{ decides } L\}.$$

We now know that HALT $\notin R$ and $\text{HALT}_\varepsilon \notin R$.

Let us now consider $\text{HALT}_\varepsilon$, and prove that it has the following property, closely related to recursivity: there is a Turing machine $\mathcal{E}$ that "enumerates" all (string representations of) machines that halt (when executed on a blank tape). $\mathcal{E}$ will run forever; we can assume that it has a dedicated output tape on which it keeps writing string representations of TMs; in particular, if $s \in \text{HALT}_\varepsilon$ then $\mathcal{E}$ will eventually write it on the output tape; is $s \notin \text{HALT}_\varepsilon$, then $\mathcal{E}$ will never write $s$ on its output tape.

```
1. procedure enumerate_halting_machines
2.     queue ← []
3.     for s in Σ*                                  Scan all strings
4.         add s to queue                           Add the new string s to queue
5.         for s′ in queue:                         Iterate on all machines in queue
6.             perform one more step of ⌊s′⌋(ε)
7.             if halting state reached then
8.                 output s′
9.                 remove s′ from queue
```

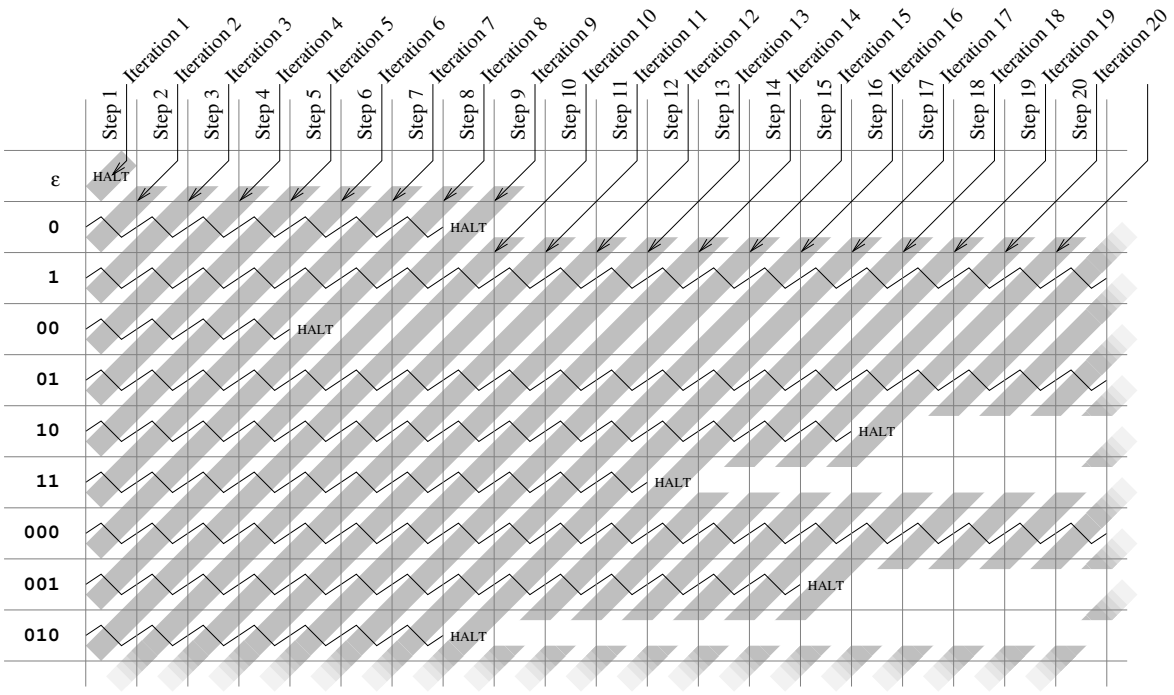



Figure 1.2: Enumerating all halting machines (on empty tape): top: the algorithm; bottom: a working schematic

The machine $\mathcal{E}$ cannot simply iterate through all strings and test them, because there is no TM that can test for $\text{HALT}_\varepsilon$. However, we can take an approach inspired to how we simulate the concurrent execution of processes in a time sharing operating system: we simulate all possible machines by alternating a step for each machine (we "dovetail" their execution); moreover, since there is an infinite number of machines, we take a diagonal approach: we create a "queue" of machines, initially empty; at every iteration we add a new machine to the queue, then we forward the execution of each machine in the queue by one step; as soon as a simulated machine halts, we remove it from the queue and put its string on the output tape. See the top part of Fig. 1.2 for a pseudocode of the procedure; the bottom part of the same figure shows the order in which all steps of all machines are executed.

**Definition 7.** *A language $L \subseteq \Sigma^*$ for which there is a TM that enumerates its elements (in the above sense) is called* recursively enumerable *(R.E.).*
*The set of all recursively enumerable languages is called RE:*

$$RE = \{L \subseteq \Sigma^* : \exists \mathcal{E} \in TM \text{ s.t. } \mathcal{E} \text{ enumerates } L\}.$$

Function $\text{HALT}_\varepsilon$ is our first example of R.E. function that is provably not recursive, i.e.:

**Theorem 9.**

$$HALT_\varepsilon \in RE \setminus R.$$

*Proof.* The method described in Fig. 1.2 and explained above defines a machine $\mathcal{E}$ that enumerates all and only halting TMs. □

Observe that a R.E. language $L$ only requires a procedure that only outputs the strings belonging to $L$, while nothing is required for strings that do not. We can also introduce the opposite definition:

**Definition 8.** *A language $L$ is* co-recursively enumerable *(co-R.E.) if there is a TM that enumerates all strings that do* not *belong to $L$. In other words, language $L$ is co-R.E. iff its complement $\bar{L} = \Sigma^* \setminus L$ is R.E.*
*The set of all co-R.E. languages is called coRE:*

$$coRE = \{L \subseteq \Sigma^* : \exists \mathcal{E} \in TM \text{ s.t. } \mathcal{E} \text{ enumerates } \Sigma^* \setminus L\}.$$

Is $\text{HALT}_\varepsilon$ also a co-R.E. language? To answer this, consider the following theorem:

**Theorem 10.** *A language is recursive if and only if it is both R.E. and co-R.E. In other words,*

$$L \in R \Leftrightarrow L \in RE \wedge L \in coRE \qquad \text{or, more succintly,} \qquad R = RE \cap coRE.$$

*Proof.* Let us prove the "only if" part first. If $L$ is recursive, then there is a TM $\mathcal{M}_L$ that decides it. But then $\mathcal{M}_L$ can be used to build an enumerating machine $\mathcal{E}$: just scan all strings $s \in \Sigma^*$, and run $\mathcal{M}_L(s)$ to decide whether to output $s$ or not. Therefore, L is R.E. The same procedure, with an inverted decision, can be used to output all strings that do not belong to $L$; hence, $L$ is also co-R.E.
Now for the "if" part: consider a language that is both R.E. and co-R.E. Since $L$ is R.E., then there is an enumerating TM $\mathcal{E}_1$ that outputs all strings in $L$; similarly, since L is co-R.E., then there is an enumerating TM $\mathcal{E}_2$ that outputs all strings that are not in $L$.
Now, suppose that we need to decide whether a string $s$ is in $L$ or not. Observe that either $\mathcal{E}_1$ or $\mathcal{E}_2$ will eventually output $s$. By dovetailing their execution, depending on which machine outputs the string, we can decide whether $s \in L$ and halt after providing the answer. □

At this point, we can conclude:

**Corollary 3.** $HALT_\varepsilon \notin coRE$.

*Proof.* We already know that $\text{HALT}_\varepsilon \in RE$. If $\text{HALT}_\varepsilon \in coRE$ were true, then we would have

$$\text{HALT}_\varepsilon \in RE \cap coRE = R,$$

but we know that $\text{HALT}_\varepsilon \notin R$. □ 2026-10-05
end

### Decision and acceptance

2026-10-06 start

Definition 7 introduces R.E. languages in terms of enumerating TMs, which start on an empty tape and output more and more elements of the language. A more practical way to introduce R.E. languages is to link them to machines that take a string as an input, as customary for recursive languages. We can prove the following alternative definitions:

**Theorem 11.** *A language $L$ is R.E. if and only if there is a TM $\mathcal{M}$ such that*

$$\forall s \in \Sigma^* \quad \mathcal{M}(s) = \begin{cases} 1 & \text{if } s \in L \\ \infty & \text{otherwise.} \end{cases} \tag{1.1}$$

*Proof.* For the "only if" part: assume that we have an enumerating TM $\mathcal{E}$ that outputs all and only strings in language $L$. Then we can define the machine $\mathcal{M}$ that, on input $s$, runs $\mathcal{E}$ step by step and keeps checking if string $s$ appears in its output. Once $s$ appears, $\mathcal{M}$ erases everything, writes "1" and halts. The machine $\mathcal{M}$ satisfies the definition, because if $s \notin L$ then $\mathcal{E}$ will never output it and $\mathcal{M}$ will never stop.
The "if" part requires the "diagonal dovetailing" toolset introduced in the proof of Theorem 9: given the TM $\mathcal{M}$ that works as described in the theorem, we define machine $\mathcal{E}$ as in Fig. 1.2 to simulate all computations $\mathcal{M}(s)$ for all strings $s \in \Sigma^*$, and print out $s$ as soon as the computation $\mathcal{M}(s)$ reaches a halting state. □

Observe that in Theorem 11 we can presume different behaviors for the machine $\mathcal{M}$, as long as it halts in an accepting state if $s \in L$. For example, the following condition, where $\mathcal{M}$ accepts strings in $L$ and does "anything else" (including running forever) otherwise, is equivalent:

$$\forall s \in \Sigma^* \quad \mathcal{M}(s) = \begin{cases} 1 & \text{if } s \in L \\ \neq 1 & \text{otherwise.} \end{cases} \tag{1.2}$$

In fact, any machine with the property 1.1 also satisfies 1.2, while given a machine with property 1.2 it can be easily modified to satisfy 1.1.

Therefore, Theorem 11 (with either condition 1.1 or 1.1) can be used as an alternate definition of R.E. languages.

From now on, we will use the following terms when speaking of languages.

**Definition 9** (accept, decide)**.** *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 *$S$.*

2026-10-06 end

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

2026-10-12 start

**Definition 10** (compute)**.** *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 12.** *A function $f : \mathbb{N} \to \mathbb{N}$ is computable if and only if there is a TM $\mathcal{M}$ on alphabet $\Sigma = \{1, \textvisiblespace\}$ 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.

### 1.4.7 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 11** (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 necessarily 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 $n \leq 5$. The current Busy Beaver candidate with $n = 6$ states writes more than $2 \uparrow\uparrow\uparrow 5$ ones before halting after more than $2 \uparrow\uparrow\uparrow 5$ steps[7].

**Theorem 13.** *The function* $S(n)$ *is not computable.*

*Proof.* Suppose that $S(n)$ is computable. Then, we could create a TM to compute $\text{HALT}_\varepsilon$ (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). Also note that the above proof follows the third scheme of Theorem 8, where the function $S(n)$ is part of the reduction.

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 14.** *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:

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

[7] In Knuth's up-arrow notation, the double arrow represents *tetration*, i.e., a tower of exponentials; until 2024 the best lower bound for $\Sigma(6)$ was

$$\Sigma(6) < 10 \uparrow\uparrow 15 = 10^{10^{10^{\cdot^{\cdot^{\cdot^{10}}}}}}$$

with 15 levels of exponentiation. Later, a 6-state machine was found to halt with more than $10 \uparrow\uparrow 11000000$ 1's on the tape (a tower of exponents with more than eleven million levels). The current contender, with $2 \uparrow\uparrow\uparrow 5$ ones, is much larger than that: the triple arrow, called *pentation*, represents repeated tetrations.

By definition, $F$ clearly has the following properties:

$$F(n) \geq f(n) \quad \forall n \in \mathbb{N}, \tag{1.3}$$

$$F(n) \geq n^2 \quad \forall n \in \mathbb{N}, \tag{1.4}$$

$$F(n+1) > F(n) \quad \forall n \in \mathbb{N} \tag{1.5}$$

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 8 at page 112);
- 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:

$$\Sigma(x+2C) \geq F(F(x)). \tag{1.6}$$

Now,

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

the first inequality comes from (1.4), 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 (1.5), we get

$$F(F(x)) >_E F(x+2C). \tag{1.7}$$

By concatenating (1.6), (1.7) and (1.3), we get

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

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

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

□

This proof is based on the original one given by Tibor Radó in 1962[8]. 2026-10-12 end

### 1.4.8 ...So what?

2026-10-13 start

What lesson can we take away from the Busy Beaver example, which is clearly purely theoretical and has no practical applications whatsoever?

We can observe that in this course we have introduce the (arguably) simplest viable computational framework (the Turing Machine), having the simplest and most objective criteria to evaluate languages and algorithms:

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

- How complex is the algorithm to solve a given problem / decide a language? Count the number of states.
- How much space does a computation require? Count the number of visited tape cells.
- How long does a computation take? Count the number of steps.

We have seen that, even in such a simple setting, behaviors that are too complex to analyze can arise in very small machines (6 states), and behaviors that are just unpredictable are not much farther ahead. This shows us that a common experience of computer scientists, the emergence of complex behavior in code, is not due to the overwhelming complexity of some computing machinery, but it is intrinsic in even the simplest systems, as soon as they are capable of computing.

## 1.5 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 $\sigma$ 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 12.** *A* property *of TMs is a decision function $\mathcal{P} : TM \to \{0, 1\}$, and we say that $\mathcal{M}$ has property $\mathcal{P}$ when $\mathcal{P}(\mathcal{M}) = 1$.*

**Definition 13.** *A property is* semantic *if its value is shared by all TMs recognizing the same language:*

$$\forall \mathcal{M}_1, \mathcal{M}_2 \in TM \quad L(\mathcal{M}_1) = L(\mathcal{M}_2) \quad \Rightarrow \quad \mathcal{P}(\mathcal{M}_1) = \mathcal{P}(\mathcal{M}_2).$$

By extension, we can say that a language $S$ has a semantic property $\mathcal{P}$ if at least one machine (and hence all machines) that recognizes $S$ has it. Finally, we define a property as *trivial* if all TMs have it, or if no TM has it — in other words, if it is constant. A property is *non-trivial* if there is at least one machine having it, and one not having it.

The two only trivial properties (the one possessed by all TMs and the one possessed 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 15** (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 on the empty input, $\text{HALT}_\varepsilon$.
Suppose that a non-trivial semantic property $\mathcal{P}$ is decidable; this means that there is a TM $\mathcal{M}_\mathcal{P}$ 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 $\mathcal{P}$ (otherwise we can work on the complementary property), and that the Turing machine $\mathcal{N}$ has the property $\mathcal{P}$ (we can always find $\mathcal{N}$ because $\mathcal{P}$ is not trivial).
Given an arbitrary machine $\mathcal{M} \in \text{TM}$, we can then check whether $\mathcal{M}(\varepsilon)$ halts or not by building the following auxiliary TM $\mathcal{M}'$ that, on input $t$, works as follows:

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

- run $\mathcal{N}$ on the original input $t$.

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

**The proof as a standard reduction**

The above proof might become clearer if restated in terms of the Turing reduction framework introduced in Section 1.4.5. In this context, the two decision functions are $f_1 = \text{HALT}_\varepsilon$ and $f_2 = \mathcal{P}$. The computable function $g$ takes machine $\mathcal{M}$ (of which we want to check the halting property) and maps it into a machine $\mathcal{M}'$ of which we want to check the property $\mathcal{P}$ in such a way that

$$\forall \mathcal{M} \in \text{TM} \quad \text{HALT}_\varepsilon(\mathcal{M}) = \mathcal{P}(\overbrace{g(\mathcal{M})}^{\mathcal{M}'})$$

or, in short,

$$\text{HALT}_\varepsilon <_T \mathcal{P},$$

i.e., $\text{HALT}_\varepsilon$ is "less uncomputable" than $\mathcal{P}$: if the latter were computable, the former would be too.

See also the Wikipedia entry for Rice's Theorem[9]. 2026-10-13 end

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

# Chapter 2

# More undecidable problems

## 2.1 Kolmogorov complexity

2026-10-20 start

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 syntax.

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 14.** *Let $\Sigma$ be a suitable alphabet (e.g., ASCII or Unicode — or just $\{0,1\}$), let $\mathcal{U}$ be a universal Turing machine working on $\Sigma$ and let $s \in \Sigma^*$ be a string. We say that the pair of strings $(m,t) \in \Sigma^* \times \Sigma^*$ is a* description *of $s$ if $m$ encodes a TM $\lfloor m \rfloor$ such that, when simulated by $\mathcal{U}$ on input $t$, produces $s$ on the tape and halts:*

$$\mathcal{U}(m,t) = \lfloor m \rfloor(t) = s.$$

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 $(m,t)$ is a *self-extracting* executable file where $m$ is a code segment contaning an algorithm that performs the decompression and $t$ is a data segment containing the compressed string on which $m$ must act upon. We usually "run" the decompression code $\lfloor m \rfloor(t)$ by double-clicking on its file's 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 $s$.

**Definition 15** (Kolmogorov complexity)**.** *Given a UTM $\mathcal{U}$ and a string $s$, we define its* Kolmogorov complexity[1] *$K_{\mathcal{U}}(s)$ to be the size of its smallest description in $\mathcal{U}$:*

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

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

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

### 2.1.1 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 16.** *Given two UTMs $\mathcal{U}$ and $\mathcal{V}$, there is a* constant *value $c_{\mathcal{UV}}$ such that, for every $s$,*

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

*Note that the constant doesn't depend on the specific string $s$.*

*Proof.* Let $s \in \Sigma^*$.
Let $(m_{\mathcal{U}}, t_{\mathcal{U}})$ be a[2] shortest description of $s$ with respect to $\mathcal{U}$ (i.e., such that $\mathcal{U}(m_{\mathcal{U}}, t_{\mathcal{U}}) = s$ and $|m_{\mathcal{U}}| + |t_{\mathcal{U}}| = K_{\mathcal{U}}(s)$).
Conversely, let $(m_{\mathcal{V}}, t_{\mathcal{V}})$ be a shortest description of $s$ in $\mathcal{V}$ (i.e., such that $\mathcal{V}(m_{\mathcal{V}}, t_{\mathcal{V}}) = s$ and $|m_{\mathcal{V}}| + |t_{\mathcal{V}}| = K_{\mathcal{V}}(s)$).
Since $\mathcal{U}$ is a UTM, it can be used to simulate $\mathcal{V}$. Let $m_{\mathcal{UV}}$ be the representation of $\mathcal{V}$ in $\mathcal{U}$. Therefore, $\big(m_{\mathcal{UV}}, (m_{\mathcal{V}}, t_{\mathcal{V}})\big)$ is a description of $s$ in $\mathcal{U}$. In fact,

$$\mathcal{U}\big(m_{\mathcal{UV}}, (m_{\mathcal{V}}, t_{\mathcal{V}})\big) = \mathcal{V}(m_{\mathcal{V}}, t_{\mathcal{V}}) = s.$$

Therefore, $|\big(m_{\mathcal{UV}}, (m_{\mathcal{V}}, t_{\mathcal{V}})\big)| \geq K_{\mathcal{U}}(x)$, and thus

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

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

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

By combining the two constants, $c_{\mathcal{UV}} = \max\{|m_{\mathcal{VU}}|, |m_{\mathcal{UV}}|\}$, we obtain the thesis. □

The theorem tells us that the specific computing substrate is not really relevant, as the size of $s$ grows, because the difference in the respective complexities remains constant.

This corresponds to having a self-extracting executable created for a specific OS, say Linux, 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 adding in front of 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.

### 2.1.2 Non-computability 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 17.** *Given the UTM $\mathcal{U}$, the Kolmogorov complexity function $K_{\mathcal{U}} : \Sigma^* \to \mathbb{N}$ is not computable.*

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

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

[2] We could write "the" shortest description, but there could be more than one of the same size: in this case any is fine.

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 $\mathcal{M}_K$ as a "subroutine", but we can safely assume that its description does not add more than a million symbols to that of $\mathcal{M}_K$, so $|\hat{\mathcal{N}}| < |\hat{\mathcal{M}}_K| + 1000000$.
- $\mathcal{N}(\varepsilon)$ is going to write a string and halt: the upper complexity bound $|\hat{\mathcal{M}}_K| + 1000000$ is *constant*, so some string must eventually overcome it.

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

$$K_{\mathcal{U}}(s) \geq |\hat{\mathcal{M}}_K| + 1000000 > |\hat{\mathcal{N}}|.$$

On the other hand, from the second point above we know that $(\hat{\mathcal{N}}, \varepsilon)$ is a description of $s$ with respect to $\mathcal{U}$, therefore

$$K_{\mathcal{U}}(s) \leq |\hat{\mathcal{N}}|,$$

therefore we have a contradiction. Observe that, if 1000000 looks too small an overhead, we can increase it as much as we want. □ 2026-10-20 end

## 2.2 "Fun facts" about computability

The following observations were discussed, in random order, during the course. While they are not required for the exam, they are reported here because their comprehension is a useful self-assessment test.

### 2.2.1 Provably undecidable machines

So we know that HALT is non-recursive: there is no TM that can decide whether a given machine $\mathcal{M}$ halts on a given input $t$, and give the correct answer for all machines $\mathcal{M}$ and all inputs $t$.
However, this leaves the following question open: "Are we aware of a *specific* TM which we are *demonstrably* incapable to decide whether it halts or not?"

Consider the following facts:

1. A formal system, such as Arithmetic, can be seen as a collection of basic truths (the *axioms*) and a few inference rules (such as the *Modus Ponens* rule, which tells us that if $A$ is true and $A \Rightarrow B$ is true then $B$ is true).
2. Starting from the list of axioms of a sound Arithmetic Theory, such as Zermelo-Fraenkel's axioms, which we assume to be true, we can systematically scan the list to find statements to which we can apply our inference rules. For instance, a pair of statements in the form "$A$" and "$A \Rightarrow B$" can be used to *prove* $B$ as follows: "Since $A$ is on the list, then it is true; the same can be said for $A \Rightarrow B$; therefore, by Modus Ponens, $B$ must be true too", so we can add $B$ to the list. Once a statement is on the list, it can be used to infer new theorems.
3. By playing this game, every theorem (i.e., provable statement) will sooner or later appear on the list.
4. This game can be played by a TM; namely, it is possible to write a TM $\mathcal{M}_{\text{ZF}}$ that writes all Zermelo-Fraenkel axioms on the tape[3] and, by repeated application of inference rules, combines

[3] There are a few additional complications: for example, Z.-F. has an infinite set of axioms; however, they belong to a finite set of parametric *axiom schemes*, and we only need to write those on the tape; every time we select an axiom scheme, we pair it with numeric values for its parameters in a systematic way

all possible known truths together to generate new true statements, that it also writes on the tape. Once a statement is on the tape, it can be used as an established truth to generate new true statements.
Every theorem in the Theory will be written down on $\mathcal{M}_{\text{ZF}}$'s tape, sooner or later.

5. Let us make $\mathcal{M}_{\text{ZF}}$ halt if and only if it generates the statement $0 = 1$; the statement is blatantly false, so every Mathematician hopes that $\mathcal{M}_{\text{ZF}}$ will *never* generate it and therefore will never halt. Note that if $\mathcal{M}_{\text{ZF}}$ generates the statement $0 = 1$ it means that it is possible to prove it; since the opposite ($0 \neq 1$) can also be proved, we would conclude that the Zermelo-Fraenkel set of axioms is inconsistent: it can prove two contradictory statements, from which every statement can be proved, no matter if true or false.

6. Here comes the problem: due to Gödel's First Incompleteness Theorem, no formal theory that can define Arithmetic (such as Zermelo-Fraenkel's set of axioms) can prove its own consistency.

7. Therefore, it's impossible to prove whether $\mathcal{M}_{\text{ZF}}$ halts (if not by making the Theory stronger by adding some specialized axiom such as "$\mathcal{M}_{\text{ZF}}$ halts" or "$\mathcal{M}_{\text{ZF}}$ never halts" to it).

Note that $\mathcal{M}_{\text{ZF}}$ isn't just a theoretical machine that one is allowed to "suppose" to exist; it is a very specific example that researchers were able to create using 745 states and a 2-symbol alphabet[4].

The existence of $\mathcal{M}_{\text{ZF}}$ sets an upper bound on the number of states for which the busy beaver function $S(n)$ is computable. In particular, it proves that $S(745)$ cannot be determined. Most researchers think, however, that $S(n)$ is uncomputable for much lower values.

**What about machines with less than 745 states?**

The largest number of states $n$ for which we know everything is $n = 5$. There are many 6-state, 2-symbol TMs for which we are still unable to decide whether they halt or not, and we don't know if the issue will be settled in the foreseeable future, if ever. Indeed, if we look at the history of the Busy Beaver problem[5], we see that while the 2- and 3-state BBs were presented in the year the problem was formulated (1964), the 4-state machine was found 10 years later, in 1974, after which 50 years passed before a collaborative effort settled the 5-state case.

However, $S(n)$ grows so fast that (theoretical) bets are being placed on a very small value of $n$ as the real "incomputability threshold" for 2-symbol TMs.

For machines with more than two symbols, the incomputability threshold might be even lower.

A very interesting starting point to explore this topic is Scott Aaronson's survey "The Busy Beaver Frontier"[6].

### 2.2.2 More consequences of the Halting Problem non-computability

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[7], the non-existence of odd perfect numbers[8]... 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 similar to the one defined in Section 2.2.1 that, starting from pre-encoded axioms, systematically generates all their consequences (theorems), but halts whenever it generates $P$. Machine $\mathcal{H}$ would tell us whether $P$ is ever going to be generated or not (i.e., whether it is provable).

[4] see for instance `https://www.ingo-blechschmidt.eu/assets/bachelor-thesis-undecidability-bb748.pdf`

[5] E.g., `https://bbchallenge.org/~pascal.michel/ha.html`

[6] `https://www.scottaaronson.com/papers/bb.pdf`

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

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

Note that, in all cases described above, we would only receive a "yes/no" answer, not an actual counterexample or a proof. However, i nthe case of a "yes" answer from $\mathcal{H}$, we could run the actual machine (being sure that it will halt) to find them.

### 2.2.3 Kolmogorov complexity, the halting problem and paradoxes

You may notice that the proof of Theorem 17 on the non-computability of Kolmogorov complexity we do not operate a reduction from the Halting problem, as was our custom up to that point. The proposed proof is in fact *independent* from it.

It is however possible to derive the non-computability of Kolmogorov complexity from that of the Halting problem and vice-versa. Finding the two proofs is indeed a good exercise. In particular, proving that the computability of HALT would imply that of $K(s)$ should be easy to figure out.

#### Berry Paradox

In proving the non-computability of Kolmogorov complexity, we search for a string $s$ of high Kolmogorov complexity (in simpler words, "that can only be obtained from a long description"), and in the process we are able to generate it with a machine ("describe it with a string") whose size is smaller than the (alleged) complexity of $s$.

This theorem is a formal rendition of the famous Berry paradox[9]:

> "The smallest positive integer that cannot be defined with less than fifteen English words."

defines such an integer with fourteen (i.e., less than fifteen) words.

### 2.2.4 Recursive enumerability of properties

Rice's Theorem puts every non-trivial semantic property on par with the Halting problem as far as computability (i.e., recursivity) is concerned. What about recursive enumerability (RE)?

Some semantic properties are clearly RE, e.g., see property $\mathcal{P}_2$ from exercise 11. Is it possible to define properties that are not RE?

The answer is yes: informally, all TM properties that require testing an infinite number of strings are not even RE. For instance,

$$\mathcal{P} = \{\mathcal{M} : \mathcal{M}(\varepsilon) \text{ accepts all words containing the substring "}\texttt{hello}\text{"}\}.$$

The precise statement of this fact is called the Rice-Shapiro Theorem, however it is out of this course's scope.

[9] `https://en.wikipedia.org/wiki/Berry_paradox`

# Chapter 3

# Complexity classes: P and NP

2026-10-26
start

From now on, we will be only dealing with computable functions; the algorithms that we will analyze 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

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

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 $n$:

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

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

## 3.2 Time classes

Let us now focus on decision problems.

**Definition 16.** *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 $f$:*

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

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.

Here are some examples of languages with their DTIME class.

**EVEN** — Strings of even length; the language can clearly be decided by a single scan of the string, i.e., by a number of steps equal to the size of the string (in "linear" time wrt the input size):

$$\text{EVEN} \in \text{DTIME}(n^1).$$

**PALINDROME** — Strings that are palindromes. On a single-tape machine, we are forced to move back and forth along the input string, removing one symbol (or group of symbols) at a time. The overall number of steps is approximately

$$T_{\mathcal{M}_{\text{PALINDROME}}}(n) = \sum_{i=0} ni = \frac{n(n-1)}{2} = O(n^2), \qquad \text{hence} \quad \text{PALINDROME} \in \text{DTIME}(n^2).$$

**SATISFIABILITY or SAT** — Given an $n$-variable Boolean expression $f(x_1, \ldots, x_n)$ (usually in conjunctive normal form, CNF[1]), is there a truth assignment to the variables that satisfies (i.e., makes true) the formula?
Although manyclever heuristics have been developed for this very important problem, they all require a significant amount of brute-force search among a large set of possible truth assignments. As the number of possible assignments on $n$ variables is $2^n$ (every variable can be either true or false), the execution time is still exponential wrt input size, so the best we can currently say is that[2]

2026-10-26 end

$$\text{SAT} \in \text{DTIME}(2^n).$$

**TRAVELING SALESPERSON PROBLEM (TSP)** — You must plan a round trip to visit all $n$ cities in a region; you know the cost of moving between each pair of cities, and you are given a budget for your travels. Is there an order of visit such that the overall traveling cost does not exceed the allowed budget[3]?

2026-10-27 start

Again, despite decades of efforts, a significant chunk of the $n!$ possible city permutations must be considered even by the smartest algorithms, so the best we can say is

$$\text{TSP} \in \text{DTIME}(n!) \subset \text{DTIME}(2^{n^2}).$$

[1] We will define it later, see Section 35; see also `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`.
[2] A slightly lower bound has been (allegedly) claimed on October 25 by an OpenAI paper, currently under review, immediately followed by a similar work by Anthropic.
[3] This will also be defined later, see Section 3.8

**Properties of DTIME($f$) classes**

Observe that, by definition, if $f = O(g)$ then DTIME($f$) $\subseteq$ DTIME($g$). Therefore, we have the following infinite hierarchy of time classes:

$$\begin{aligned} \text{DTIME}(n) \subseteq \text{DTIME}(n^2) \subseteq \quad \cdots \quad \subseteq \text{DTIME}(n^k) \subseteq \cdots \\ \cdots \subseteq \text{DTIME}(2^n) \subseteq \text{DTIME}(2^{n^2}) \subseteq \quad \cdots \quad \subseteq \text{DTIME}(2^{n^k}) \subseteq \cdots \end{aligned} \tag{3.1}$$

In particular, every time class in the first row is dominated by a time class of thesecond row.

Also, observe that if we allow multiple-tape Turing machines then PALINDROME can be decided in linear time, therefore PALINDROME $\in$ DTIME($n$). Hence, the precise value of $k$ in DTIME($n^k$) might depend on the computational model that we use. In particular, we already know from Theorem 3 that any $k$-tape Turing machine can be simulated by a single-tape TM with at most a quadratic slowdown. In other words, if a language is in DTIME($n^k$) for a multiple-tape TM, it is in DTIME($n^{2k}$) for single-tape machines.

### 3.2.1 Polynomial-time and exponential-time languages

We aren't going to be interested in the precise exponent $k$ of the above definitions. Therefore, let us define the set of all languages that are decidable within time bounded by *some* power (i.e., by some polynomial function) of the input size:

**Definition 17.**

$$\boldsymbol{P} = \bigcup_{k=1}^{\infty} \mathit{DTIME}(n^k).$$

*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 $x$*

$$(x \in L \Leftrightarrow \mathcal{M}(x) = 1) \quad \wedge \quad t_{\mathcal{M}}(x) \leq p(|x|). \tag{3.2}$$

This definition is independent of the precise computational model. In fact, even though the precise exponent $k$ depends on the machine we use, **P** contains *all* possible exponents.

We can do the same with exponential time:

**Definition 18.**

$$\boldsymbol{EXP} = \bigcup_{k=1}^{\infty} \mathit{DTIME}(2^{n^k}).$$

*In other words, we say that a language $L \in \Sigma^*$ is* exponential-time*, and write $L \in \boldsymbol{EXP}$, if there are a machine $\mathcal{M}$ and an exponent $k$ such that for every input string $x$*

$$(x \in L \Leftrightarrow \mathcal{M}(x) = 1) \quad \wedge \quad t_{\mathcal{M}}(x) \leq p(|x|). \tag{3.3}$$

Observe that **P** is in some sense the limit (closure) of the first row of (3.1), and **EXP** is the closure of the second row. Therefore, we get the first of many inclusion theorems that we will find in this course:

**Theorem 18.**

$$\boldsymbol{P} \subseteq \boldsymbol{EXP}.$$

Table 3.1: Growth of runtimes for algorithms of different complexities (columns), as the input size grows (rows); time is normalized by the runtime of the $n = 10$ case.

| Input | Compute time at different complexity classes | | | | | |
|---|---|---|---|---|---|---|
| size $n$ | DTIME($n$) | DTIME($n^2$) | DTIME($n^3$) | DTIME($2^n$) | DTIME($n!$) | DTIME($2^{n^2}$) |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 |
| 11 | 1.1 | 1.21 | 1.331 | 2 | 11 | $2^{21}$ |
| 12 | 1.2 | 1.44 | 1.728 | 8 | 132 | $2^{44}$ |
| 13 | 1.3 | 1.69 | 2.197 | 16 | 1716 | $2^{69}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 20 | 2 | 4 | 8 | 1024 | $\approx 6 \cdot 10^{11}$ | $2^{300}$ |

### 3.2.2 P is good and EXP is bad

To motivate definitions 17 and 18, consider Table 3.2.2 where we write down the fictitious runtimes of six equally fictitious algorithms of increasing time complexity (columns) as we increase the input size from $n = 10$ to $n = 20$. To simplify things, since we are only concerned about how times *grow*, let's assume that the runtime for $n = 10$ is always 1 time unit. In other words, the table shows how much longer must an algorithm of a given complexity take if we increase the input size.

Our main motivation for wishing a language to be polynomial-time is the enormous discrepancy between the growth rate of the three columns on the left (referring to **P** languages) and the three on the right (**EXP** languages): the first three look treatable, the other three show that the problem quickly becomes unaffordable.

#### Is P always treatable?

However, we should point out two issues with this clear-cut division of the world in "good" and "bad" time classes.

**Galactic algorithms** — Time classes are defined in asymptotic terms: in particular, the big-O notation gets rid of multiplicative constants. This means that a supposedly treatable language in DTIME($n^3$) might have an actual time bound of $10^{100}n^3$ time steps. In this case, an exponential algorithm running in time $10^{-3}2^n$ still works for reasonable values of $n$, and the supposedly better cubic algorithm only becomes competitive for unrealistically large values of $n$. The Wikipedia page on so-called Galactic algorithms[4] contains many examples.

**High-degree polynomials** — The polynomial degree can be a problem. Although $n^{1000}$ is still a polynomial, it grows so fast that, again, its advantage over an exponential time $2^n$ only becomes apparent for very large values of $n$.

**Quasi-polynomial time** — Conversely, although $n^{\log n}$ grows faster than any polynomial, because its exponent is not constant and keeps growing, the exponent growth is so slow that it might as well be treated as a polynomial function for "reasonable" values of $n$. Complexity theorists define the class

$$\mathbf{QP} = \bigcup_{k=1}^{\infty} \mathrm{DTIME}(n^{(\log n)^k}) = \bigcup_{k=1}^{\infty} \mathrm{DTIME}(2^{(\log n)^k}).$$

Thus, **QP** is an intermediate class of languages, harder than **P** and easier than **EXP**:

$$\mathbf{P} \subseteq \mathbf{QP} \subseteq \mathbf{EXP}.$$

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

### 3.2.3 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 $x_1, \ldots, x_n$, we can define the following[5]:

- 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 19** (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*

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

*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:

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) \quad = \quad & (x_1 \vee \neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \\ & \wedge (\neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5). \end{aligned} \tag{3.4}$$

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 (3.4):

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

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

Note that CNF is powerful enough to express any (non-quantified) 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 $y$."

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, specifically:

- Commutativity:

$$\begin{aligned} a \vee b \quad &\equiv \quad b \vee a \\ a \wedge b \quad &\equiv \quad b \wedge a; \end{aligned}$$

- Distribution:

$$\begin{aligned} a \vee (b \wedge c) \quad &\equiv \quad (a \vee b) \wedge (a \vee c) \\ a \wedge (b \vee c) \quad &\equiv \quad (a \wedge b) \vee (a \wedge c); \end{aligned}$$

- De Morgan laws:

$$\begin{aligned} \neg(a \vee b) \quad &\equiv \quad \neg a \wedge \neg b \\ \neg(a \wedge b) \quad &\equiv \quad \neg a \vee \neg b; \end{aligned}$$

- Implication:

$$a \to b \quad \equiv \quad \neg a \vee b.$$

[5] See also `https://en.wikipedia.org/wiki/Conjunctive_normal_form`

## 3.3 NP languages

While two of the languages listed above (SAT and TSP) cannot be decided by any known polynomial-time algorithm, they share a common property: if a string is in the language[6], there is an "easily" (polynomial-time) verifiable proof of it:

- If $f(x_1, \ldots, x_n) \in$ SAT (i.e., boolean formula $f$ is satisfiable), then there is a truth assignment to the variables $x_1, \ldots, x_n$ that satisfies it. If we were given this truth assignment, we could easily check that, indeed, $f \in$ 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 a list of cities and travel costs is satisfiable within a certain budget (therefore it belongs to TSP), in order to verify this all we need is the order of visit of the cities, which can be easily checked by verifying that all cities are touched and that the sum of costs does not exceed the budget.

2026-10-27 end 2026-11-02 start

In other words, a computational system able to afford exponential-time algorithms could convince us of a positive answer by providing a *certificate* (or *witness*), by means of which 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 20.** *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 $T_{\mathcal{M}} \leq p$ (i.e., $\mathcal{M}$ always halts in polynomial time wrt its input size) and, for every input string $x$,*

$$x \in L \quad \Leftrightarrow \quad \exists c \in \Sigma^{q(|x|)} : \mathcal{M}(x, c) = 1. \tag{3.5}$$

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.

Here are a few more languages belonging to the **NP** time class:

**CLIQUE** — Given an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}$, does $G$ have a clique (i.e., a complete subgraph) of size $k$?

**INDSET (INDEPENDENT SET)** — Given an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}$, does $G$ have an independent set (i.e., a subset of mutually disconnected nodes) of size $k$?

**SUBSET SUM** — Given a finite set of numbers $S \subset \mathbb{N}$ and a target sum $t \in \mathbb{N}$, is there a subset $S' \subseteq S$ whose sum is precisely $\sum S' = t$?

All of these languages have quite obvious exponential-time algorithms (test all subsets until one meets the requirements); even the smartes branch-and cut heuristics cannot exclude enough cases to reduce the complexity to polynomial time. As is often the case, we are unable to rule out the possibility that a polytime algorithm exists, but we are far from having found one.

[6] Remember, when we say "a string is in SAT" we mean that the string represents a satisfiable Boolean formula; likewise, "a string in in TSP" means that the string represents a set of cities and a maximum budget, and that a round trip within the given budget is possible.

**Exponential languages that are unlikely to be NP**

Let us clarify that not all **EXP** languages seem to share this "polytime verifiability" property.

For instance, consider the REDUCED HALT language:

$$\text{REDUCED HALT} = \{(\mathcal{M}, k \in \mathbb{N}) \mid \mathcal{M}(\varepsilon) \text{ halts in less than } k \text{ steps}\}.$$

This language is clearly computable, but there must be some pairs $(\mathcal{M}, k)$ for which we need to simulate $\mathcal{M}(\varepsilon)$ for $k$ steps. The time required by the simulation is $O(|\hat{\mathcal{M}}|k)$ (i.e., $k$ times the simulation of a step, which in turn requires scanning the macine's representation). However, the input size is just $O(|\hat{\mathcal{M}}| + \log k)$, since we can write $k$ in positional (i.e., binary) notation. Therefore, if we set $n = \log k$, the execution time is $O(|\hat{\mathcal{M}}|2^n)$; hence,

$$\text{REDUCED HALT} \in \text{DTIME}(|\hat{\mathcal{M}}|2^n) \subseteq \mathbf{EXP}.$$

However, we have no idea of a polynomial witness (although, again, we haven't ruled out the possibility). Therefore, it is unlikely that the language belongs to **NP**.

Other problems in game theory, such as the analysis of positions in the generalized $(n \times n)$ versions of Chess, Checkers and Go, belong to the same category: they are in **EXP** because the analysis requires visiting large portions of the exponential move trees, and no short certificates for winning positions are known.

### 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 21** (Non-deterministic Turing machine)**.** *A non-deterministic Turing Machine (NDTM) is a TM with two 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 16 with a NDTM:

**Definition 22.** *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 $f$:*

2026-11-02 end

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

2026-11-03 start

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 19.**

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

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

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

Let $L \in \mathbf{NP}$, as in Definition 20. 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 20 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 $x \notin L$ 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 20, 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 $x \notin L$, 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 20 hold, and $L \in \mathbf{NP}$. □

### 3.3.2 An asymmetry in the definition of NP: the class coNP

Observe that the definition of **NP** introduces an asymmetry in acceptance and rejection that is reminiscing of the asymmetry between RE and coRE languages. Namely, while we require only one accepting computation to accept $x \in L$, in order to reject it we require that *all* computations reject it.

This means that, while $x \in L$ admits a polynomial certificate, and therefore is verifiable even by a deterministic polynomial checker, the opposite $x \notin L$ does not: there is no hope for a polynomial checker to become convinced that $x \notin L$.

**Definition 23.** *The symmetric class to **NP** is called **coNP**: the class of languages that have a polynomially verifiable certificate for strings that do* not *belong to the language.*

$$\mathbf{coNP} = \{L \subseteq \Sigma^* : \bar{L} \in \mathbf{NP}\}.$$

As an example, consider SAT: given a boolean CNF formula $f : \{0,1\}^n \to \{0,1\}$, is there a truth assignment that satisfies $f$? As far as we know, there is no way to produce a certificate that polynomially verifies a negative answer.

Given a CNF formula $f$, its negation is easily represented by a disjunction of conjunctive clauses (a formula in *Disjunctive Normal Form*, DNF), as we can perform a repeated application of De Morgan's laws (invert all $\vee$'s and $\wedge$'s, then negate all terms — remembering that double negations cancel out):

$$\begin{aligned} \neg f(x_1, \ldots, x_n) &= \neg \bigwedge_{i=1}^{m} \bigvee_{j=1}^{l_i} g_{ij} \\ &= \bigvee_{i=1}^{m} \neg \bigvee_{j=1}^{l_i} g_{ij} \\ &= \bigvee_{i=1}^{m} \bigwedge_{j=1}^{l_i} \neg g_{ij}. \end{aligned}$$

Asking whether $f$ is satisfiable is equivalent to asking if $\neg f$ is a tautology (i.e., always satisfied), then reversing the answer. We can therefore define the language:

**TAUTOLOGY** — Given a Boolean DNF formula $f$, is $f$ satisfied by *all* truth assignments?

**Theorem 20.** *TAUTOLOGY* $\in$ ***coNP***.

*Proof.* The above discussion is enough, since we have shown that $f \in \text{TAUTOLOGY} \Leftrightarrow \neg f \notin \text{SAT}$; however, the discussion can be summarized in a more direct argument: if $f$ is *not* a tautology, then it can be polynomially verified by a truth assignment that falsifies it, while clearly such truth assignment cannot be provided if $f$ is always satisfied. □

We can define **coNP** in terms of non-deterministic TMs as follows. Remember that, up to now, we defined acceptance of input $x$ by a NDTM $\mathcal{N}$ as the existence of at least one accepting computation in the trace of $\mathcal{N}(x)$. We can define this "acceptance mode" of a NDTM as the "existential mode," and add a new one where acceptance is required by all computation paths:

**Definition 24.** *An* existential-mode *NDTM (in short, $\exists$-NDTM or simply NDTM) is a non-deterministic TM $\mathcal{N}^{\exists}$ which is said to accept its input $x$ iff the non-deterministic computation $\mathcal{N}^{\exists}(x)$ has at least one accepting branch*[8]*.*
*Conversely, a* universal-mode *NDTM (in short $\forall$-NDTM) is a non-deterministic TM $\mathcal{N}^{\forall}$ which is said to accept its input $x$ iff* all *branches of the non-deterministic computation $\mathcal{N}^{\forall}(x)$ are accepting.*

Therefore, a $\forall$-NDTM $\mathcal{N}^{\forall}$ rejects an input $x$ iff at least one branch of $\mathcal{N}^{\forall}(x)$ ends in rejection.

We now have a symmetrical landscape: definition 23 mirrors definition 20, and universal-mode NDTMs "mirror" existential-mode NDTMs, so that the following result can be proved by, again, mirroring the proof of Theorem 19:

**Theorem 21.** ***coNP*** *is the class of languages that are decided in polynomial time by some $\forall$-NDTM.*

Clearly, given that a deterministic TM is just a special case of NDTM with a single-valued transition function, the following is true:

**Theorem 22.**

$$\boldsymbol{P} \subseteq \boldsymbol{NP} \cap \boldsymbol{coNP}.$$

As usual, we don't know whether the inclusion is strict or is an equality. 2026-11-03 end

## 3.4 Reductions and hardness

2026-11-09 start

Nobody knows if **NP** is a proper super-set 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 25.** *Given two languages $L, L' \in$ **NP**, we say that $L$ is* polynomially many-one 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 $|x|$.*

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 $L \leq_{\text{P}} L'$, 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.

[8] Again, this is what we usually refer to as NDTM, unless specified otherwise.

### 3.4.1 Simple examples

**Reductions between versions of SAT**

**Definition 26** ($k$-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, (3.4) 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 27.** *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 23.** *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 a non-satisfiable formula. □

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 24.**

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

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

$$\bigvee_{j=1}^{l_i} g_{ij} \tag{3.6}$$

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

$$\left( h \vee \bigvee_{j=1}^{l_i - 2} g_{ij} \right) \wedge (\neg h \vee g_{i,l_i-1} \vee g_{il_i}), \tag{3.7}$$

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 (3.6) also satisfy (3.7), and vice versa. In fact, if (3.6) is satisfied then at least one of its literals are true; but then one of the two clauses of (3.7) 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 (3.7) are satisfied, then at least one of the literals in (3.6) 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. □

As an example, the 4-CNF formula (3.4) can be reduced to the following 3-CNF with the two additional variables $h$ and $k$ used to split its two 4-clauses:

$$\begin{aligned} f'(x_1, x_2, x_3, x_4, x_5, h, k) \quad = \quad & (h \vee x_1 \vee \neg x_2) \wedge (\neg h \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \\ & \wedge (k \vee \neg x_1 \vee \neg x_2) \wedge (\neg k \vee x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5). \end{aligned} \tag{3.8}$$

Theorem 24 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 23, 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 and INDEPENDENT SET (INDSET for short). The two problems are almost the same, but in the second 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

$$\text{CLIQUE} \leq_{\text{P}} \text{INDSET} \qquad \text{and} \qquad \text{INDSET} \leq_{\text{P}} \text{CLIQUE}.$$

### 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 25.**

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

*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 (3.8): 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 $f$. □

### 3.4.3 Another example: reducing SAT to an arithmetic problem

Here is another example of reduction between two languages from different "domains" (Boolean formulas and integer inequalities). First of all, let us define the following language:

**INTEGER LINEAR PROGRAMMING (ILP)** — Given a set of $m$ linear inequalities with integer coefficients on $n$ integer variables, is there at least a solution? In other terms, given $n \times m$ coefficients $a_{ij} \in \mathbb{Z}$ and $m$ bounds $b_i \in \mathbb{Z}$, does the following set of inequalities

$$\begin{cases} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & \leq & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & \leq & b_m \end{cases}$$

have a solution with $x_1, \dots, x_n \in \mathbb{Z}$?

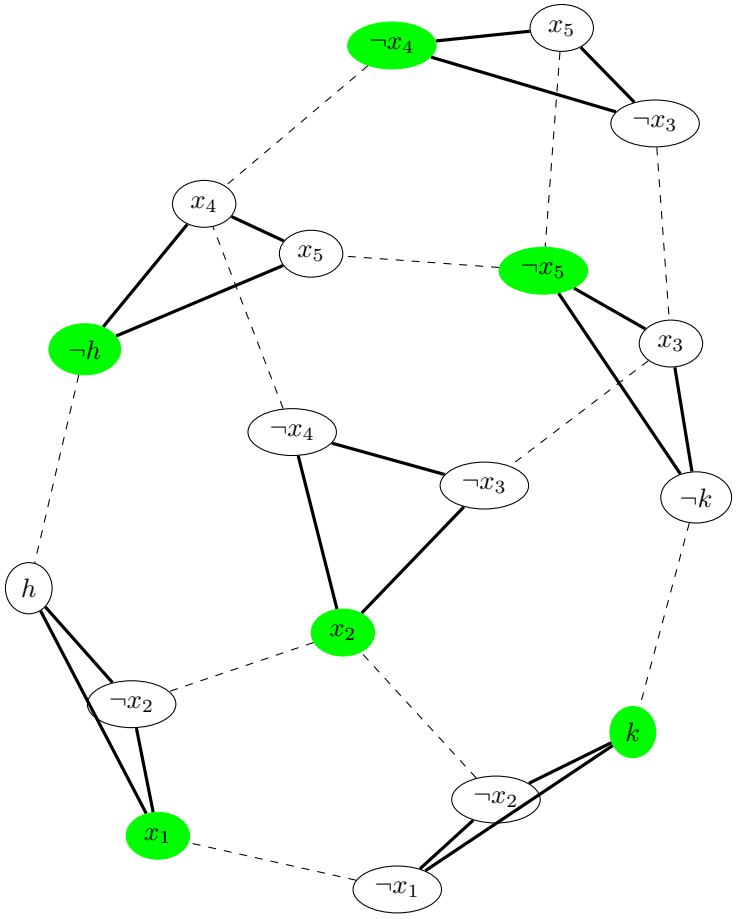



Figure 3.1: Reduction of the 3-CNF formula (3.8) to a graph for INDSET.

ILP is clearly **NP**, because an integer assignment to $x_1, \ldots, x_n$ can be verified with a polynomial number of additions and multiplications. Moreover, no solving polynomial-time algorithm is known. Let's show that ILP is at least as hard as SAT by showing a polynomial reduction, i.e., a SAT instance can be easily transformed into an equivalent ILP instance:

**Theorem 26.**

$$SAT \leq_P ILP.$$

*Proof.* Let $f$ be a CNF formula with variables $x_1, \ldots, x_n$. We can directly map them to $n$ variables of an ILP.

The first constraint is that every variable must be 0 or 1; this can be translated into two constraints per variable:

$$-x_i \leq 0, \quad x_i \leq 1 \qquad \text{for } i = 1, \ldots, n;$$

notice that, in order to use "$\leq$" inequalities $x_i \geq 0$ has been reformulated by changing the sign. Next, every clause must be true. We can translate this into one constraint per clause, where we require that the sum of the literals that compose it is not zero. Literal $x_i$ is mapped onto itself, while a negated literal $\neg x_i$ can be translated into the "arithmetic" equivalent of negation $1 - x_i$. For instance, clause $(x_3 \vee \neg x_9 \vee x_{16})$ is rendered into

$$x_3 + (1 - x_9) + x_{16} \geq 1, \qquad \text{i.e.,} \qquad -x_3 + x_9 - x_{16} \leq 0.$$

Therefore, a CNF formula with $n$ variables and $m$ clauses is mapped onto a ILP problem with $n$ variables and $2n + m$ constraints. 2026-11-09

□ end

## 3.5 NP-hard and NP-complete languages

2026-11-10 start

In this Section we will introduce the most relevant result in this branch of Complexity Theory; namely, that all **NP** languages can be polynomially reduced to 3-SAT. Informally, this makes 3-SAT the "hardest" of all **NP** languages, because if we were able to solve it in polynomial time then every **NP** language would be solvable in polynomial time too, and we would have $\mathbf{P} = \mathbf{NP}$.

To do so, we will first introduce logic gates and Boolean circuits as a practical way of describing Boolean dependencies; then we will show how a Boolean circuit can be transformed into a 3-CNF formula (Tseytin transformation); then we'll see how a non-deterministic Turing machine can be expressed as a Boolean circuit and, finally, how a full computation by a NDTM can be represented by a Boolean circuit. By putting all of the above together, we will see how acceptance of an input by a NDTM can be expressed as the satisfiability of a 3-CNF formula, and how all of this can be carried out in polynomial time.

### 3.5.1 Boolean circuits and 3-CNF formulas: the Tseytin transformation

Our first point of order will be the following: suppose that we are given a generic Boolean formula with "and", "or", and "not" operators, and that we need to convert it into CNF form. By applying Boolean expression rules, we are guaranteed that it is possible; however, we might end up with an exponentially larger formula. For example, let' s see the worst case, a $k$-DNF formula (a *disjunctive* normal form formula with $k$ terms per clause) with $m$ clauses:

$$\begin{aligned} f(x_1, \ldots, x_n) &= \bigvee_{i=1}^{m} \bigwedge_{j=1}^{k} t_{ij} \\ &= (t_{11} \wedge t_{12} \wedge \cdots \wedge t_{1k}) \vee (t_{21} \wedge t_{22} \wedge \cdots \wedge t_{2k}) \vee \cdots \vee (t_{m1} \wedge t_{m2} \wedge \cdots \wedge t_{mk}). \end{aligned}$$

Direct conversion into CNF requires the repeated application of distribution of disjunction wrt conjunction, resulting in the conjunction of all disjunctive clauses that can be formed by picking a term from the first clause (of the original formula), a term from the second, and so on until the $m$-th clause:

$$\begin{aligned} f(x_1, \ldots, x_n) &= \bigwedge_{i_1,\ldots,i_m=1,\ldots,1}^{k,\ldots,k} \bigvee_{j=1}^{m} t_{j,i_j} \\ &= (t_{11} \vee t_{12} \vee \cdots \vee t_{1m}) \wedge (t_{21} \vee t_{12} \vee \cdots \vee t_{1m}) \wedge \cdots \wedge (t_{k1} \vee t_{12} \vee \cdots \vee t_{1m}) \wedge \ldots \\ &\quad \cdots \wedge (t_{11} \vee t_{22} \vee \cdots \vee t_{1m}) \wedge (t_{21} \vee t_{22} \vee \cdots \vee t_{1m}) \wedge \cdots \wedge (t_{k1} \vee t_{22} \vee \cdots \vee t_{1m}) \wedge \ldots \\ &\quad \vdots \\ &\quad \cdots \wedge (t_{11} \vee t_{k2} \vee \cdots \vee t_{km}) \wedge (t_{21} \vee t_{k2} \vee \cdots \vee t_{km}) \wedge \cdots \wedge (t_{k1} \vee t_{k2} \vee \cdots \vee t_{km}). \end{aligned}$$

This CNF formula is equivalent to the original DNF, it still has $n$ variables, but with $m$ terms per clause and $k^m$ clauses, therefore it is exponentially larger, while we need to have at most polynomial increases in size.

Another, equivalent 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 operators, together with their truth tables and CNF formulas that are satisfiable by all truth assignments that are compatible with the gate. These

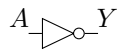
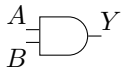
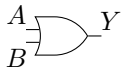
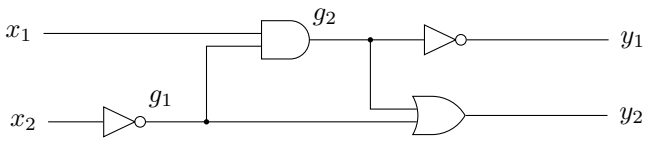
| Gate | Truth table | Tseytin transformation |
|---|---|---|
|   | $\begin{array}{c\|c} A & Y \\ \hline 1 & 0 \\ 0 & 1 \end{array}$ | $\begin{aligned} & Y = \neg A \\ \equiv\ & (Y \Rightarrow \neg A) \wedge (\neg A \Rightarrow Y) \\ \equiv\ & (\neg Y \vee \neg A) \wedge (Y \vee A) \end{aligned}$ |
|   | $\begin{array}{cc\|c} A & B & Y \\ \hline 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{array}$ | $\begin{aligned} & Y = A \wedge B \\ \equiv\ & \big(Y \Rightarrow (A \wedge B)\big) \wedge \big((A \wedge B) \Rightarrow Y\big) \\ \equiv\ & \big(\neg Y \vee (A \wedge B)\big) \wedge \big(Y \vee \neg(A \wedge B)\big) \\ \equiv\ & (\neg Y \vee A) \wedge (\neg Y \vee B) \wedge (Y \vee \neg A \vee \neg B) \end{aligned}$ |
|   | $\begin{array}{cc\|c} A & B & Y \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}$ | $\begin{aligned} & Y = A \vee B \\ \equiv\ & \big(Y \Rightarrow (A \vee B)\big) \wedge \big((A \vee B) \Rightarrow Y\big) \\ \equiv\ & \big(\neg Y \vee (A \vee B)\big) \wedge \big(Y \vee \neg(A \vee B)\big) \\ \equiv\ & (\neg Y \vee A \vee B) \wedge \big(Y \vee (\neg A \wedge \neg B)\big) \\ \equiv\ & (\neg Y \vee A \vee B) \wedge (Y \vee \neg A) \wedge (Y \vee \neg B) \end{aligned}$ |

Figure 3.2: A NOT gate (top), an AND gate (middle) and an OR gate (bottom), their truth tables, and derivations of their Tseytin transformation, i.e., the 3-CNF formulas 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 3-CNF representation: the 3-CNF formula is satisfiable by precisely the combinations of truth values that are compatible with the logic gates.

formulas are known as the *Tseytin transformation*[9] of the gates: they represent the constraints that the input and output variables must obey in order to be compatible with the gate. Note that the Tseytin transformations are 3-CNF and are composed by at most 3 clauses each.

A Boolean circuit is composed by $n$ input variables (or lines, or wires) $x_1, x_2, \ldots, x_n$ and by some number $l$ of gates $G_1, G_2, \ldots, G_l$; the inputs of each gate are connected to the output of some gate or to the input variables; finally, a number $m$ of gate outputs is designated as outputs $y_1, y_2, \ldots, y_m$ of the Boolean circuit.

We only consider *combinatorial* Boolean circuits, i.e., circuits that do not preserve states: there are no "feedback loops", and gates $G_1, \ldots, G_l$ are ordered so that every gate $G_i$ only receives inputs from "previous" gates $G_j$, with $j < i$.

The Tseytin transformation allows us to "translate" any combinational Boolean circuit 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 $x_1, \ldots, x_n$, $m$ outputs $y_1, \ldots, y_m$ and $l$ gates $G_1, \ldots, G_l$:

- label every wire connected to an input $x_i$ or to an output $y_j$ with the input or output variable; for the remaining wires, observe that each is connected to the output of some gate $G_i$: assign the label $g_i$ to it;
- once all gate inputs and outputs have been assigned a variable, write the conjunction of all CNF formulas from Fig. 3.2 for all gates, replacing placeholders $A$, $B$ and $Y$ with the wire labels.

Fig. 3.3 shows an example: a Boolean circuit with 2 inputs $(x_1, x_2)$, 2 outputs $(y_1, y_2)$ and 2 ancillary variables $(g_1, g_2)$ associated to intermediate gates, together with the corresponding CNF formula. This formula completely expresses the dependency between all variables in the circuit, and by replacing truth assignment we can use it to express various questions about the circuit in terms of satisfiability. For example:

1. Is there a truth assignment to inputs $x_1, x_2$ such that the outputs are both 0?
   We can reduce this question to 3-SAT by replacing $y_1 = y_2 = 0$ (and, of course, $\neg y_1 = \neg y_2 = 1$) in the CNF of Fig. 3.3, and by simplifying we obtain

   $$(x_2 \vee g_1) \wedge (\neg x_2 \vee \neg g_1) \wedge (\neg g_2 \vee x_1) \wedge (\neg g_2 \vee g_1) \wedge (g_2 \vee \neg x_1 \vee \neg g_1) \wedge (g_2) \wedge (\neg g_1) \wedge (\neg g_2),$$

   which is clearly not satisfiable because of the conjunction $g_2 \wedge \neg g_2$.

[9] `https://en.wikipedia.org/wiki/Tseytin_transformation`

2. If we fix $x_1 = 1$, is it possible (by assigning a value to the other input) to get $y_2 = 1$?
   To answer this let us replace $x_1 = y_2 = 1$ and $\neg x_1 = \neg y_2 = 0$ into the CNF and simplify:

$$(x_2 \vee g_1) \wedge (\neg x_2 \vee \neg g_1) \wedge (\neg g_2 \vee g_1) \wedge (g_2 \vee \neg g_1) \wedge (y_1 \vee g_2) \wedge (\neg y_1 \vee \neg g_2) \wedge (g_1 \vee g_2).$$

   The formula is satisfiable by $x_2 = y_1 = 0$, $g_1 = g_2 = 1$, so the answer is "yes, just set the other input to 0".

Note that in this second case we can "polynomially" verify that the CNF is satisfiable by replacing the values provided in the text. In general, on the other hand, verifying the unsatisfiability of a CNF can be hard, because we cannot provide a certificate.

Observe that the Tseytin transformation of a Boolean circuits employs at most 3 clauses and 1 additional variable per gate, therefore the resulting formula has a size that is polynomial wrt the size of the Boolean circuit, and also its construction (labeling wires and writing clauses) is polynomial.

### 3.5.2 Using Boolean circuits to express Turing Machine computations

Now that we are able to transform any Boolean circuit into a 3-CNF Boolean formula of linear size, let's focus on describing a TM as a Boolean circuit.

As an example, consider the following machine with 2 symbols $(0, 1)$ and 2 states $(s_1, s_2)$ plus the halting state, with the following transition table:

| | 0 | 1 |
|---|---|---|
| $s_1$ | $1, s_1, \rightarrow$ | $1, s_2, \leftarrow$ |
| $s_2$ | $0, s_1, \leftarrow$ | $0, \text{HALT}, \rightarrow$ |

Suppose that we want to implement a Boolean circuit that, receiving the current tape symbol and state as an input, provides the new tape symbol, the next state and direction as an output. We can encode all inputs of this transition table in Boolean variables as follows:

- the input, being in $\{0, 1\}$, already has a canonical Boolean encoding, let us call it $x_1$;
- the two states can be encoded in a Boolean variable $x_2$ with an arbitrary mapping, for instance:

$$0 \mapsto s_1, \qquad 1 \mapsto s_2.$$

The outputs, that encode the entries of the transition table can be similarly mapped:

- the new symbol on the tape is, again, a Boolean variable $y_1$;
- the new state requires two bits, because we need to encode the HALT state. Therefore, we will need an output $y_2$ that encodes the continuation states as before, and an output $y_3$ that is true when the machine must halt. Therefore, the mapping from $y_2, y_3$ to the new state is

$$00 \mapsto s_1, \qquad 10 \mapsto s_2, \qquad 01 \mapsto \text{HALT},$$

  with the combination $y_2 = y_3 = 1$ left unused;
- the direction is arbitrarily mapped on the output variable $y_4$ ,e.g.,

$$0 \mapsto \leftarrow, \qquad 1 \mapsto \rightarrow .$$

Fig. 3.4 shows the Boolean circuit that outputs the new machine configuration (encodings of state, symbol and direction) based on the current (encoded state and symbol) pair.

2026-11-10
end

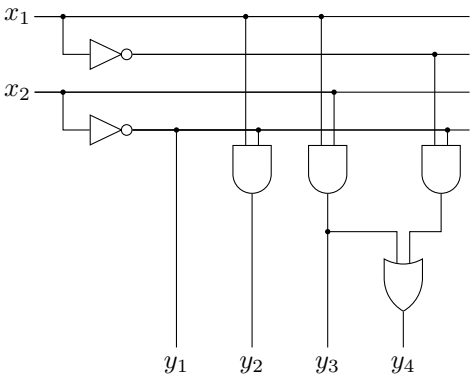



Figure 3.4: The Boolean circuit that implements the transition table of the TM described in the text.

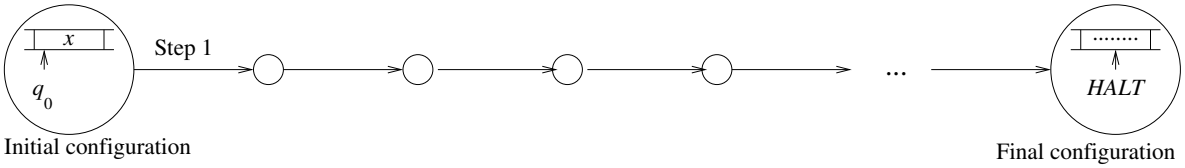



Figure 3.5: A TM computation described as a sequence of configurations, linked by steps (each consisting of the application of a transition rule). The initial configuration corresponds to the tape containing the initial input $x$, the initial current position and the initial state $q_0$. The sequence might be infinite or terminate with a configuration in a halting state, as in the example.

**Computations as sequences of machine configurations**

2026-11-16
start

In order to proceed, we need to describe the computation of a deterministic TM as a sequence of "configurations", i.e., snapshots of all information needed to describe it:

**Definition 28** (configuration)**.** *A "configuration" (or "snapshot") of a TM consists of three pieces of information:*

- *the content of the tape,*
- *the current position, and*
- *the current state.*

A computation is a sequence of configurations, starting from the initial one and possibly ending with a halting state, as shown in Fig. 3.5. The transition from a configuration to the next one is dictated by the TM's transition function.

The above example suggests that a step of a Turing machine can be executed by a circuit, and that by concatenating enough copies of this circuit we obtain a circuit that executes a whole TM computation:

**Lemma 1.** *Let $\mathcal{M}$ be a polynomial-time machine whose execution time on inputs of size $n$ is bounded by polynomial $p(n)$. Then there is a polynomial $P(n)$ such that for every input size $n$ there is a Boolean circuit $C$, whose size (in terms, e.g., of number of gates) bound by $P(n)$, that performs the computation of $\mathcal{M}$.*

*Proof outline.* Let $\mathcal{M}$ have $|Q| = m$ states. Let us fix the input size $n$. Then we know that $\mathcal{M}$ halts within $p(n)$ steps. Since every step changes the current position on the tape by one cell, the machine

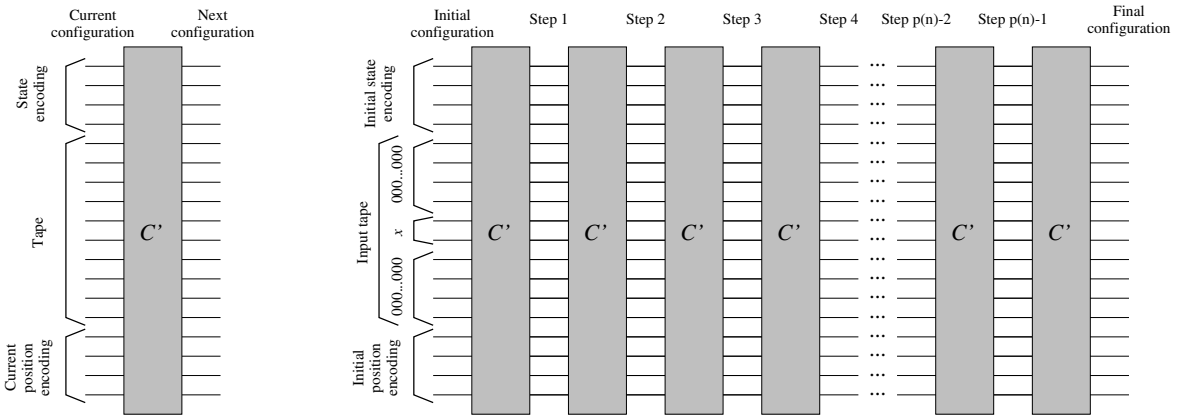



Figure 3.6: (left) $C'$ is a Boolean circuit with a polynomial number of inputs, gates and outputs with respect to the size of the TM's input $x$. It transforms a Boolean representation of a configuration of the TM into the configuration of the subsequent step. (right) By concatenating $p(|x|)$ copies of $C'$, we get a polynomial representation of the whole computation of the TM on input $x$.

will never visit more than $2p(n) + 1$ cells (considering the two extreme cases of the machine always moving in the same direction). The complete configuration of the machine at a given point in time is therefore described by:

- $2p(n) + 1$ boolean variables (bits) to describe the content of the relevant portion of the tape;
- $|Q|$ bits to describe the state;
- $2p(n) + 1$ bits to describe the current position on the tape (one of the bits is 1, the others are 0).

Of course, more compact representations are possible, e.g., by encoding states and positions in base-2 notation. By using building blocks such as the transition table circuit of Fig. 3.4, we can actually build a Boolean circuit $C'$ that accepts as an input the configuration of $\mathcal{M}$ at a given step and outputs the new configuration; this circuit has a number of inputs, outputs and gates that are polynomial with respect to $n$.

By concatenating $p(n)$ copies of $C'$ (see Fig. 3.6), we compute the evolution of $\mathcal{M}$ for enough steps to emulate the execution on any input of size $n$. By inserting the initial configuration on the left-hand side, the circuit outputs the final configuration.

If the size of every block $C'$ is bound by polynomial $q(n)$, then the size of the whole circuit is bound by $P(n) = p(n) \cdot q(n)$, therefore it is still polynomial. □

Note that the proof is not complete: in particular, the size of $C'$ is only suggested to be polynomial, but we would need to look much deeper in the structure of $C'$ to be sure of that.

**Lemma 2.** *Lemma 1 also works if the TM is non-deterministic.*

*Proof outline.* Let us recall the definition of a deterministic TM as a sequence of configurations, as shown in Section 3.5.2, and in particular in Fig. 3.5. A non-deterministic computation can be likewise represented as a directed graph where every configuration is linked to one or more "successors" depending on the number of non-deterministic steps it can take (in our case two, without loss of generality). In other words, at every step the computation "splits" in two branches, and our aim is to find a sequence of branch choices that leads to an accepting state (if any), as shown in Fig. 3.7. In order to carry on a specific branch of a NDTM's computation, we just need to modify the circuit $C'$ of Lemma 1 to accept one more input bit, and use it to choose between the two possible outcomes of the transition table as shown in Fig. 3.8. Let us call $x_1, \ldots, x_{p(n)}$ the additional input bits of the

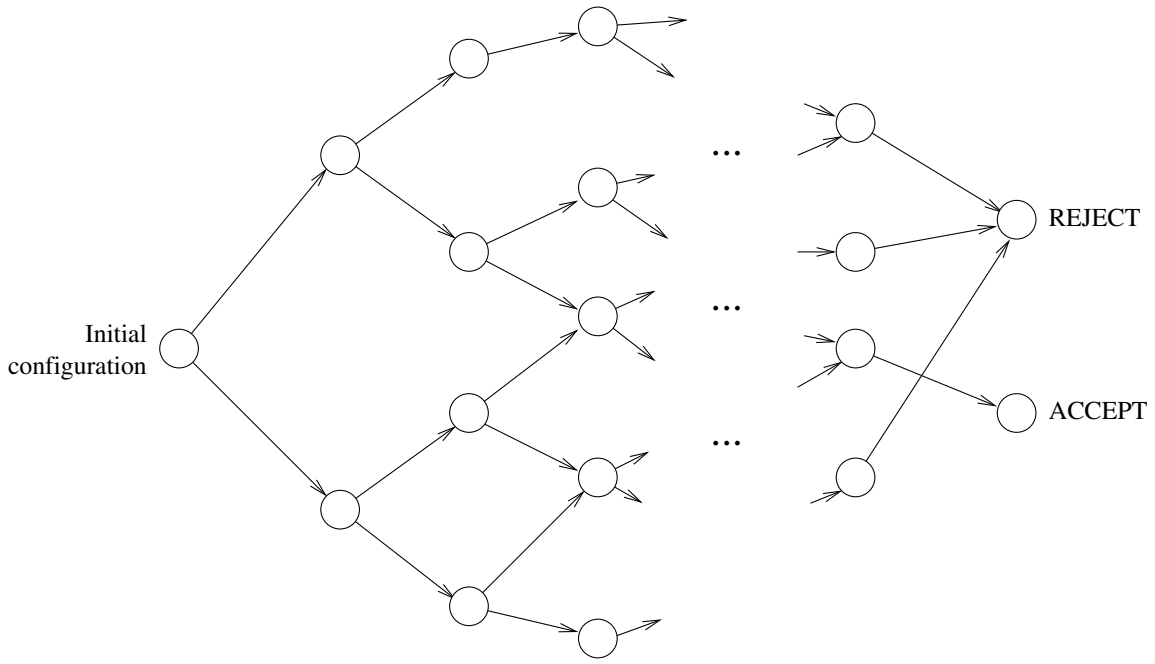



Figure 3.7: A NDTM computation described as a directed graph of configurations, linked by steps (each consisting of the application of a non-deterministic transition rule). In the case we are considering, the initial configuration corresponds to the input tape containing the initial input $x$, an empty working tape, the initial current positions and the initial state $q_0$. As we assume that every computation halts, the graph is acyclic and all paths terminate to an accepting or a rejecting state, as in the example.

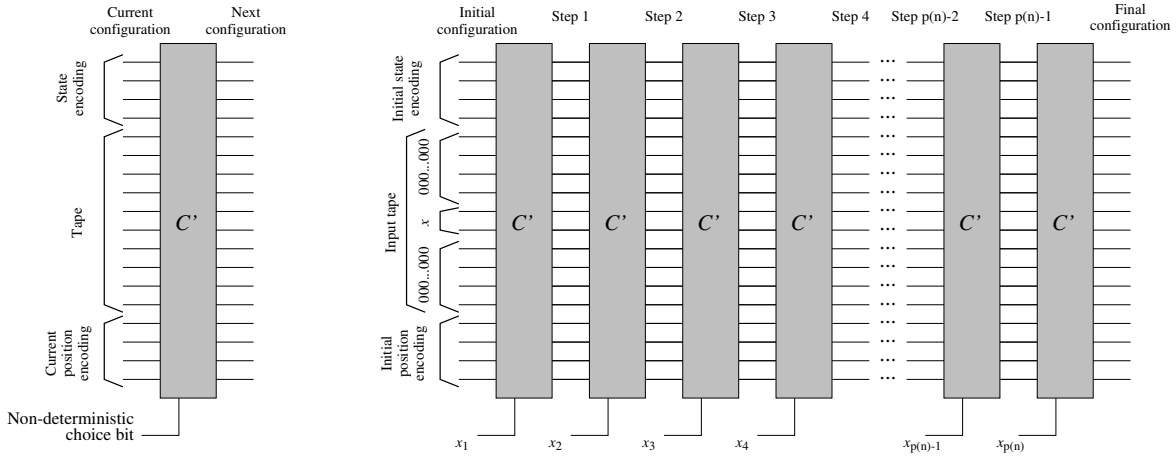



Figure 3.8: Analogous to Fig. 3.6 for a NDTM. (left) Every $C'$ block has an additional input that allows the selection of the non-deterministic choice for the step that it controls. (right) The whole circuit has $p(n)$ additional Boolean inputs $x_1, \ldots, x_{p(n)}$: every combination of choice bits represents one of the $2^{p(n)}$ computation paths of the NDTM.

daisy-chained $C'$ blocks. Each of the $2^{p(n)}$ combinations of these inputs determines one of the possible computation paths of the NDTM. □

We have thus shown how to reduce any language in $L \in \mathbf{NP}$ to a Boolean circuit whose output represents one of the final configurations of the NDTM that computes $L$, including the final (accepting or rejecting) state, depending on $p(n)$ choice bits.

Knowing this, we can see how any polynomial computation of a NDTM can be represented by a CNF formula that is only satisfiable if the NDTM accepts its input.

Consider the following definition:

**Definition 29.** *A language $L$ is said to be* **NP**-hard *if for every language $L' \in$ **NP** we have that $L' \leq_P L$.*

We are finally ready to prove the main result:

**Theorem 27** (Cook-Levin Theorem)**.** *3-SAT is **NP**-hard.*

*Proof outline.* To prove this, we need to pick a generic language $L \in \mathbf{NP}$ and show that $L \leq_{\mathrm{P}}$ 3-SAT.

Let $\mathcal{N}$ be the NDTM that decides $x \in L$ within the polynomial time bound $p(|x|)$.

Let $x \in \Sigma^n$ be a string of length $n$. By Lemma 2, we can build a Boolean circuit $C$ with polynomial size that, for any truth value combination of the inputs $x_1, \ldots, x_{p(n)}$, performs one of the $2^{p(n)}$ computations of $\mathcal{N}$.

We can transform the Boolean circuit $C$ into a (still polynomial-size) CNF formula $f_C$ by means of the procedure outlined in Sec. 3.5.1.

At this point, the question whether $x \in L$ or not, which can be expressed as "is there at least one computation of $\mathcal{N}(x)$ that ends in an accepting state?", can be answered by assigning the proper truth values to some variables in $f_C$:

- the "initial state" inputs are set to the representation of the initial state;
- the "input tape" inputs are set to the representation of string $x$ on $\mathcal{N}$'s tape;
- the "initial position" inputs are set to the representation of $\mathcal{N}$'s initial position on the tape;
- the variables corresponding to the "final state" outputs are set to the representation of the accepting halting state.

After simplifying for these preset values, the resulting CNF formula $f'_C$ still has a lot of free variables, among which are the choice bits $x_1, \ldots, x_{p(n)}$.

By construction, the CNF formula $f'_C$ is satisfiable if and only if there is a computation of $\mathcal{N}$ that starts from the initial configuration with $x$ on the tape and ends in an accepting state. Therefore,

$$x \in L \quad \leftrightarrow \quad f'_C \in \text{3-SAT}.$$

□

Thus, we have shown 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, it is straightforward that 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 30.** *A language $L \in$ **NP** that is **NP**-hard is said to be* **NP**-complete.

Of course, we already know that 3-SAT $\in$ **NP**, hence by the Cook-Levin theorem and the above definition:

**Corollary 4.** *3-SAT is **NP**-complete.*

## 3.6 More NP-complete languages

**NP**-complete languages have an important role in complexity theory: they provide an upper bound for how hard can a language in **NP** be.

Since the composition of two polynomial-time reductions is still a polynomial-time reduction, we have the following:

**Theorem 28.** *If $L$ is **NP**-hard and $L \leq_P L'$, then $L'$ is **NP**-hard too.*

So, whenever we reduce an **NP**-complete language to any other language $L \in \mathbf{NP}$, we can conclude that $L'$ is **NP**-complete too.

From Theorem 23, and from the fact that SAT $\in$ **NP**, we get the following immediate consequence:

**Corollary 5.** *SAT is **NP**-complete.*

Next, from Theorem 25, and from the fact that INDSET $\in$ **NP**, we get:

**Corollary 6.** *INDSET is **NP**-complete.*

Likewise, from Theorem 26 and the fact that ILP $\in$ **NP** we get

**Corollary 7.** *ILP is **NP**-complete.*

We have already established the equivalence between INDSET and CLIQUE, therefore

**Corollary 8.** *CLIQUE is **NP**-complete.*

Let us introduce a few more problems in **NP**.

**$k$-VERTEX COLORING** — Given an undirected graph $G = (V, E)$, is there an assignment from $V$ to $\{1, \ldots, k\}$ ("$k$ colors") such that two connected vertices have different colors?

In particular, let's consider the case $k = 3$, 2026-11-16 end

**Theorem 29.** *3-VERTEX COLORING is **NP**-complete.*

*Proof.* Let's start from a 3-CNF formula $f$ and build a graph that is 3-colorable if and only if $f$ is satisfiable.

The graph will be composed of separate "gadgets" (subgraphs) that capture the semantics of a 3-CNF formula: the construction can be followed in Fig. 3.9.

The first gadget is a triangle whose nodes will be called $T$ ("true"), $F$ ("false") and $B$ ("base"). Among the three colors, the one that will be assigned to node $T$ will be considered to correspond to assigning the value "true" to a node. Same for $F$. The three nodes are used to "force" specific values upon other nodes of the graph.

The second set of gadgets is meant to assign a node to every literal in the formula. For every variable $x_i$, there will be two nodes, called $x_i$ and $\neg x_i$. Since we are interested to assigning them truth values, we connect all of them to node $B$, so that they are forced to assume either the "true" or the "false" color. Furthermore, we connect node $x_i$ to $\neg x_i$ to force them to take different colors.

Next, every 3-literal clause is represented by an OR gadget whose "exit" node is forced to have color "true" by being connected to $B$ and to $F$. The three "entry" nodes of the gadget are connected to the nodes corresponding to the clause's literals. We can easily verify that every OR gadget is 3-colorable if and only if at least one of the literal nodes it is connected to is not false-colored.

By construction, if $f$ is a satisfiable 3-CNF formula, then it is possible to color the literal nodes so that every OR gadget has at least one true-colored node at its input, and therefore the graph will be colorable. If, otherwise, $f$ is not satisfiable, then every coloring of the literal nodes will result in an OR gadget connected to three false-colored literals, and therefore will not be colorable. □

The following language should be much simpler:

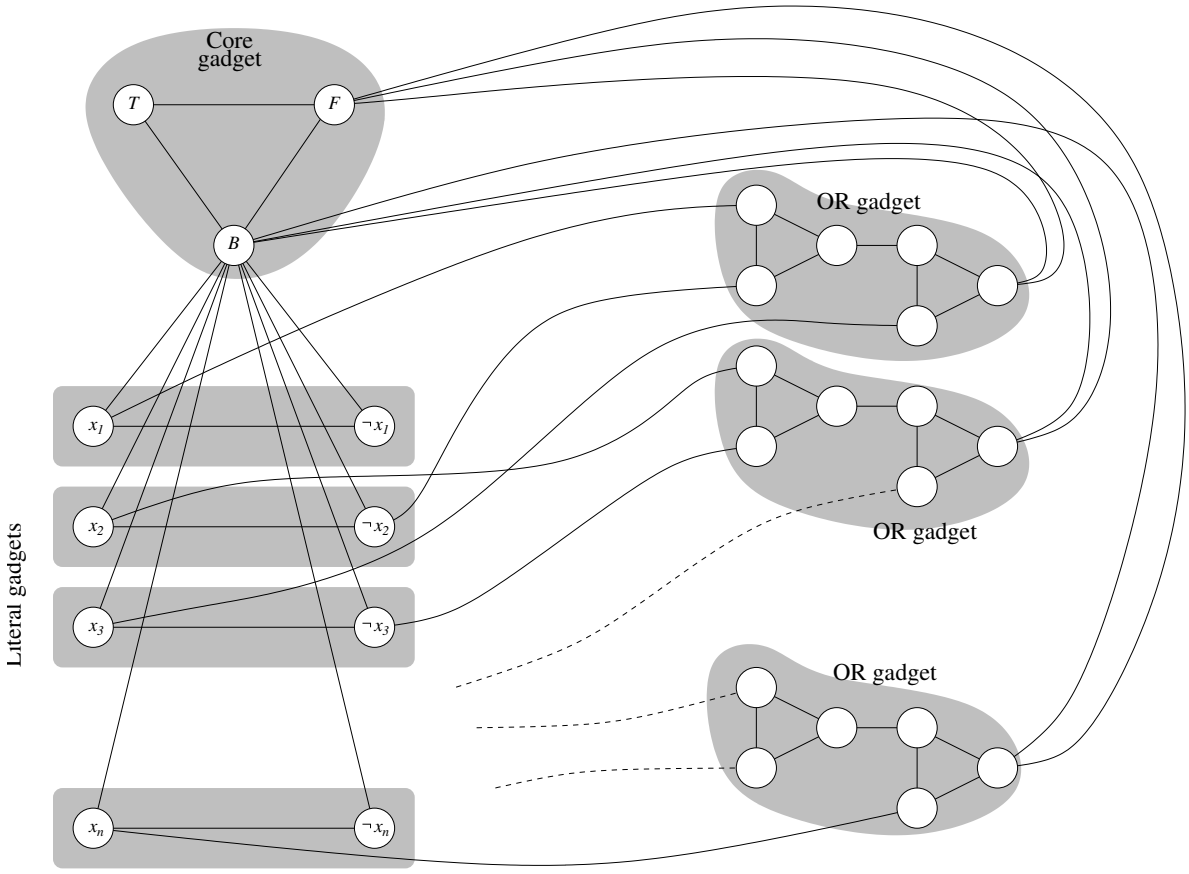



Figure 3.9: Reduction of a 3-CNF formula to the VERTEX COLORING problem with $k = 3$ colors.

**VERTEX COVER** — Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, is there a vertex subset $V' \subseteq V$ of size (at most) $k$ such that every edge in $E$ has at least one endpoint in $V'$?

**Theorem 30.** *VERTEX COVER is **NP**-complete.*

*Proof.* First of all, VERTEX COVER $\in$ **NP**, since a cover $V'$ of size $k$ is polynomially-sized wrt the problem instance and is verifiable in polynomial time.

Observe that if $V'$ of size $k$ is an independent set in $G = (V, E)$, then its complement $V \setminus V'$ is a vertex cover of size $|V| - k$ and vice versa. □

Here is a combinatorial problem about sets:

**Definition 31** (SET COVER)**.** *Given a finite set $S$, $n$ subsets $C_1, \ldots, C_n \subseteq S$ and an integer $k \in \mathbb{N}$, is it possible to select $k$ subsets $C_{i_1}, C_{i_2}, \ldots, C_{i_k}$ such that their union is $S$?*

**Theorem 31.** *SET COVER is **NP**-complete.*

*Proof.* First, SET COVER is clearly in **NP**.

In order to prove completeness, we start from VERTEX COVER. Given a graph $G = (V, E)$, let $S = E$ in the SET COVER definition, and map every vertex $i \in V$ to set

$$C_i = \{e \in E : i \in e\}$$

of all edges that have vertex $i$ as an endpoint. Solving SET COVER for $k$ subsets amounts to finding $k$ subsets (vertices of $G$) such that every element of $S$ (every edge of $G$) belongs to at least one of them (has an endpoint in one of these vertices). □

In other words, we view every vertex as the set of its edges, and we redefine the relation "$v$ is an endpoint of $e$" as "$v$ contains $e$".

## 3.7 Arithmetic problems

We already know a bunch of **NP**-complete languages about logic (e.g., SAT) and graphs (e.g., CLIQUE). The only problem about arithmetic that we have met until now is ILP. Two more problems are worth mentioning due to their extensive real-world applications: SUBSET SUM and KNAPSACK.

### 3.7.1 SUBSET SUM

**Definition 32** (SUBSET SUM)**.** *Let $w_1, w_2, \ldots, w_n \in \mathbb{N}$, and let $s \in \mathbb{N}$. The problem asks if there is a subset $I \subseteq \{1, \ldots, n\}$ such that*

$$\sum_{i \in I} w_i = s. \tag{3.9}$$

*Or, equivalently, is there a subset of indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that $\sum_{j=1}^{k} w_{i_j} = s$?*
*Or, again, is there an $n$-bit string $(b_1, b_2, \ldots, b_n) \in \{0, 1\}^n$ such that $\sum_i b_i w_i = s$?*

**Theorem 32.** *SUBSET SUM is **NP**-complete.*

*Proof.* As always, let us start by observing that SUBSET SUM $\in$ **NP**. In fact, the input size is $n + 1$ times the representation of the largest of the numbers (plus, possibly, a representation of $n$ itself), therefore it is $|x| = O(n \log \max\{x_1, \ldots, x_n, s\})$. A suitable certificate is a list of indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, of size $O(k \log n)$, which is linearly bounded by $|x|$. Checking the certificate requires $k \leq n$ sums of $(\log s)$-bit numbers, which is again linearly bounded by the input size.

In order to prove that SUBSET SUM is **NP**-hard, let us reduce 3-SAT (which we know to be **NP**-hard) to it. Let $F$ be a 3-CNF boolean formula with $n$ variables and $m$ clauses. For every variable $x_i$ in $F$, let us build two numbers with the following base-10 representation:

$$\begin{aligned} t_i &= a_1 a_2 \cdots a_n p_1 p_2 \cdots p_m, \\ f_i &= a_1 a_2 \cdots a_n q_1 q_2 \cdots q_m, \end{aligned} \tag{3.10}$$

where the digits of the numbers $t_i$ and $f_i$ are:

$$\begin{aligned} a_j &= \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise,} \end{cases} \\ p_j &= \begin{cases} 1 & \text{if the } j\text{-th clause contains } x_i \text{ without negation} \\ 0 & \text{otherwise,} \end{cases} \\ q_j &= \begin{cases} 1 & \text{if the } j\text{-th clause contains } \neg x_i \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{3.11}$$

For instance, consider the following $n = 3$-variable, $m = 4$-clause formula:

$$F(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

The formula corresponds to the following $n + 3$ pairs of $m + n$-digit numbers:

| | | $a_1$ | $a_2$ | $a_3$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|---|
| $t_1$ | = | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| $f_1$ | = | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| $t_2$ | = | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| $f_2$ | = | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| $t_3$ | = | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| $f_3$ | = | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

Note that, although the digits are 0 or 1, the numbers are represented in base 10 (e.g., $t_1$ reads "one million, one thousand, one hundred and one"). The three leftmost digits of each number act as indicators of the variable the number refers to, while the four rightmost ones identify the clauses that would be satisfied if $x_i$ were true (in the case of $t_i$) or if $x_i$ were false (in the case of $f_i$).

Let a truth assignment to $x_1, \ldots, x_n$ correspond to the choice of one number between each $t_i, f_i$ pair; namely, let us choose $t_i$ if the corresponding $x_i$ is assigned to be true, $f_i$ otherwise. For example, the truth assignment $(x_1, x_2, x_3) = (\top, \bot, \top)$ in the example corresponds to the choice of numbers $t_1$, $f_2$ and $t_3$. Observe that the sum of the three numbers is

$$t_1 + f_2 + t_3 = 1112203$$

The digits of the sum tell us that, for each variable $x_i$, exactly one number between $t_i$ and $f_i$ has been chosen (the leftmost $n$ digits are 1), and that the four clauses are satisfied by respectively 2, 2, 0 and 3 of their literals. In particular, we get the information that the third clause of $F$ is not satisfied. On the other hand, the assignment $(\bot, \bot, \top)$ corresponds to the choice of variables $f_1$, $f_2$ and $t_3$, whose sum is $f_1 + f_2 + t_3 = 1111112$, so that we know that all clauses are satisfied by at least one of their literals.

We can conclude that $F$ has a satisfying assignment if and only if a subset of the corresponding numbers $t_i, f_i$ can be found whose sum is in the form

$$s = \overbrace{11\cdots1}^{n \text{ digits}} s_1 s_2 \cdots s_m, \qquad \text{with } s_1, \ldots, s_m \neq 0. \tag{3.12}$$

In order to obtain a proper instance of SUBSET SUM, we need to transform (3.12) into a precise value. Note that, as every clause has at most 3 literals, $s_j \leq 3$. Therefore, we need to provide enough numbers to enable all non-zero $s_j$'s to become precisely 3. We can obtain this by declaring two equal numbers $u_i, v_i$ per clause, with all digits set to zero with the exceptions of the $n+i$-th digit equal to one:

$$u_i = v_i = \overbrace{00\cdots0}^{n \text{ digits}} d_1 d_2 \cdots d_m, \tag{3.13}$$

with

$$d_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

. Therefore, $F$ has a satisfying truth assignment if and only if we can find a subset among the numbers $t_1, \ldots, t_n, f_1, \ldots, f_n, u_1, \ldots, u_m, v_1, \ldots, v_m$ as defined in (3.10), (3.11), (3.13), (3.14), whose sum is

$$s = \overbrace{11\cdots1}^{n \text{ digits}} \overbrace{33\cdots3}^{m \text{ digits}}.$$

□

### 3.7.2 KNAPSACK

A simple but very important extension of SUBSET SUM is the following, where *two* sets of numbers are involved.

**Definition 33** (KNAPSACK)**.** *Given a set of $n$ items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, a knapsack with capacity $c$ and a minimum value $s$ that we want to carry, is there a subset of items that the knapsack would be able to carry and whose overall value is at least $s$?*
*More formally, the problems asks if there is a subset of indices $I \subseteq \{1, \ldots, n\}$ such that*

$$\sum_{i \in I} w_i \leq c \qquad \textit{and} \qquad \sum_{i \in I} v_i \geq s. \tag{3.15}$$

The first constraint ensures that the knapsack is not going to break, the second one ensures that we can pack at least the desired value.

Observe that KNAPSACK $\in$ **NP**, since the subset $I$ is smaller that the problem size and the sums can be verified in a comparable number of steps, so that $I$ is a suitable certificate.

Moreover, SUBSET SUM is a special case of KNAPSACK: given $(x_1, \ldots, x_n, s)$ as in definition 32, we can reformulate (hence reduce) it as a KNAPSACK instance by letting $w_i = v_i = x_i$ and $c = s$ (so, equating weights and values), which would reduce (3.15) to the equality (3.9). Therefore,

**Theorem 33.** *KNAPSACK is **NP**-complete.*

## 3.8 Problems on graphs: paths and traveling salesmen

Finally, let us consider another important problem.

**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 "cost" $c_{ij}$) and a "budget" $B$, is there an order of visit (permutation) $\pi$ of all nodes such that

$$\left( \sum_{i=1}^{n-1} w_{\pi_i \pi_{i+1}} \right) + w_{\pi_n \pi_1} \leq B, \tag{3.16}$$

i.e., the total cost along the corresponding closed path in that order of visit (also considering return to the starting node) is within budget[10]?

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

It is clear that the problem is in **NP**: a certificate is the permutation $\pi$: we can check that $\pi$ is indeed a permutation of the first $n$ integers and that the total cost of the associated path is within budget. This problem continuously appears in logistics applications, therefore the notion that it is "at least as hard" as any other problem in **NP** has quite negative implications for the real world. To prove completeness, we shall proceed by steps.

### 3.8.1 Hamiltonian paths

Our first step is to categorize the type of path that the TSP requires on a graph. We will first consider directed graphs.

**Definition 34.** *Given a directed graph $G = (V, E)$, a path in $G$ is called* Hamiltonian *if it touches every node in $V$ exactly once.*

The computational problem that we want to consider is the following:

**Definition 35** (HAMILTONIAN PATH)**.** *Given a directed graph $G = (V, E)$ and two distinct nodes $s, t \in V$, is there a Hamiltonian path in $G$ starting from $s$ and ending in $t$?*

Unsurprisingly:

**Theorem 34.** *HAMILTONIAN PATH is **NP**-complete.*

*Proof.* Clearly, HAMILTONIAN PATH $\in$ **NP**: a certificate is the path itself, expressed as a list on nodes, which can be easily checked for the desired properties: $s$ is the first node, $t$ is the last, every node in $V$ appears exactly once, two consecutive nodes are connected by an edge in the correct direction.

Let us consider a reduction from SAT; i.e., given a generic CNF expression, let us create a graph that has a Hamiltonian path between two specified nodes if and only if the expression is satisfiable.

Let $f$ be a CNF expression on $n$ variables $x_1, \ldots, x_n$ organized as the conjunction of $n$ disjunctive clauses $C_1, \ldots, C_m$. The main structure of the corresponding graph $G$ is a chain of $n$ "diamonds", one for each variable, as in the left side of Fig. 3.10, with $s$ at the top and $t$ at the bottom. The middle chain of every diamond is doubly linked, and can be traversed by a path in either direction. For this reason, it should be clear that the graph has a multitude of Hamiltonian paths, and we can encode a simple correspondence between the truth assignment to a variable and the direction of traversal of its horizontal chain; e.g., let us assume that a left-to-right traversal corresponds to assigning "true", and right-to-left means "false". Figure 3.10 (right) shows one such Hamiltonian path and the corresponding truth assignment to variables.

Next, we add a node for each clause $C_1, \ldots, C_m$, and we encode the relationship between variables and clauses as follows; consider the equation

$$\overbrace{(x_1 \vee \cdots)}^{C_1} \wedge \overbrace{(\neg x_1 \vee x_2 \vee \cdots)}^{C_2} \wedge \cdots \wedge \overbrace{(\neg x_2 \vee \cdots \neg x_n)}^{C_m}$$

with reference to Fig. 3.11:

- Every horizontal chain has a consecutive pair of nodes for every clause, each pair separated from the neighboring ones and from the diamond edges by an additional "buffer" node.
- If clause $C_i$ contains the *positive* literal $x_j$, then we add an edge from the left node in the appropriate pair in the horizontal chain of $x_j$ to node $C_i$, and an edge from $C_i$ to the right node of the pair. For example, since $x_1$ appears in clause $C_1$, we connect the third node in the chain of $x_1$ to node $C_1$, and back from $C_1$ to the fourth node of the chain. This connection allows for a "detour" when traversing the chain from left to right, but not vice versa.
- If clause $C_i$ contains the *negative* literal $\neg x_j$, then we connect the same two nodes in the opposite order; for example, since $\neg x_1$ appears in clause $C_2$, we connect the seventh node of $x_1$'s chain to $C_2$, and $C_2$ back to the sixth node. This allows for a "detour" only when traversing the chain from right to left.

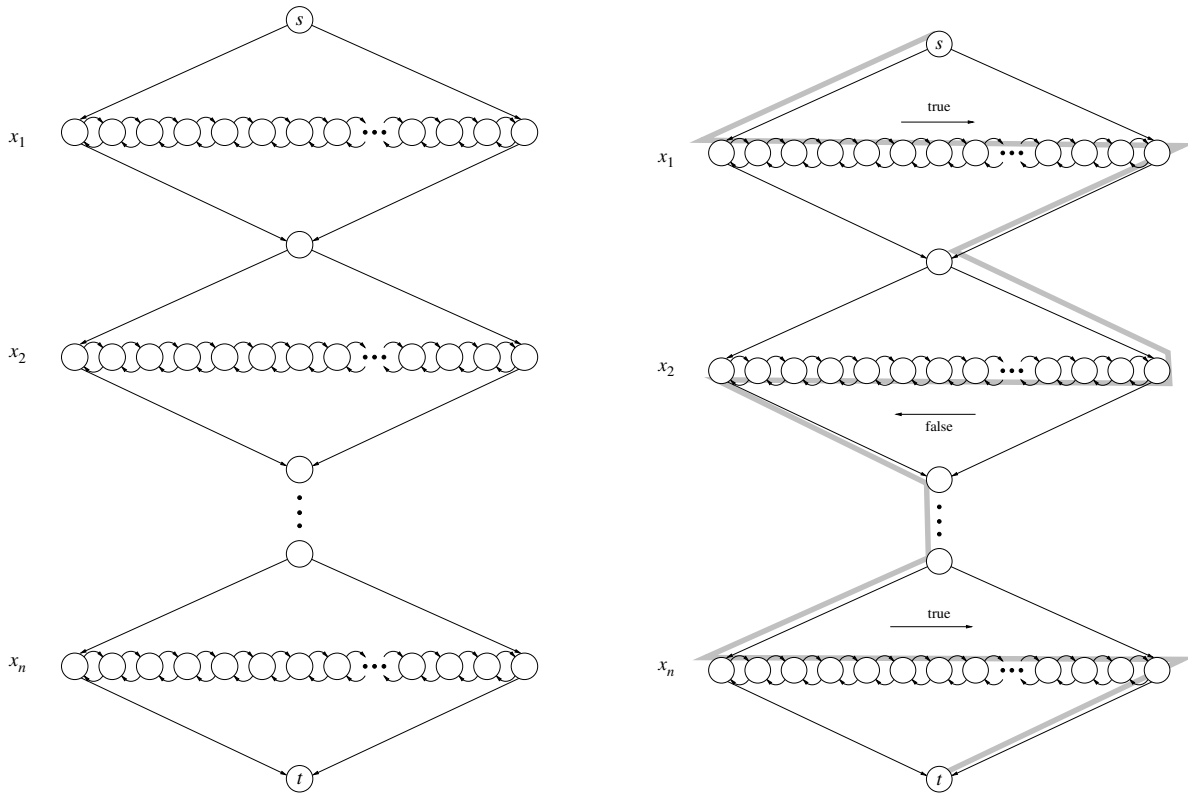



Figure 3.10: Left: directed graph representing Boolean variables $x_1, \ldots, x_n$. Right: Hamiltonian path from $s$ to $t$ encoding a truth assignment to variables ($x_1 = x_n = \top$, $x_2 = \bot$).

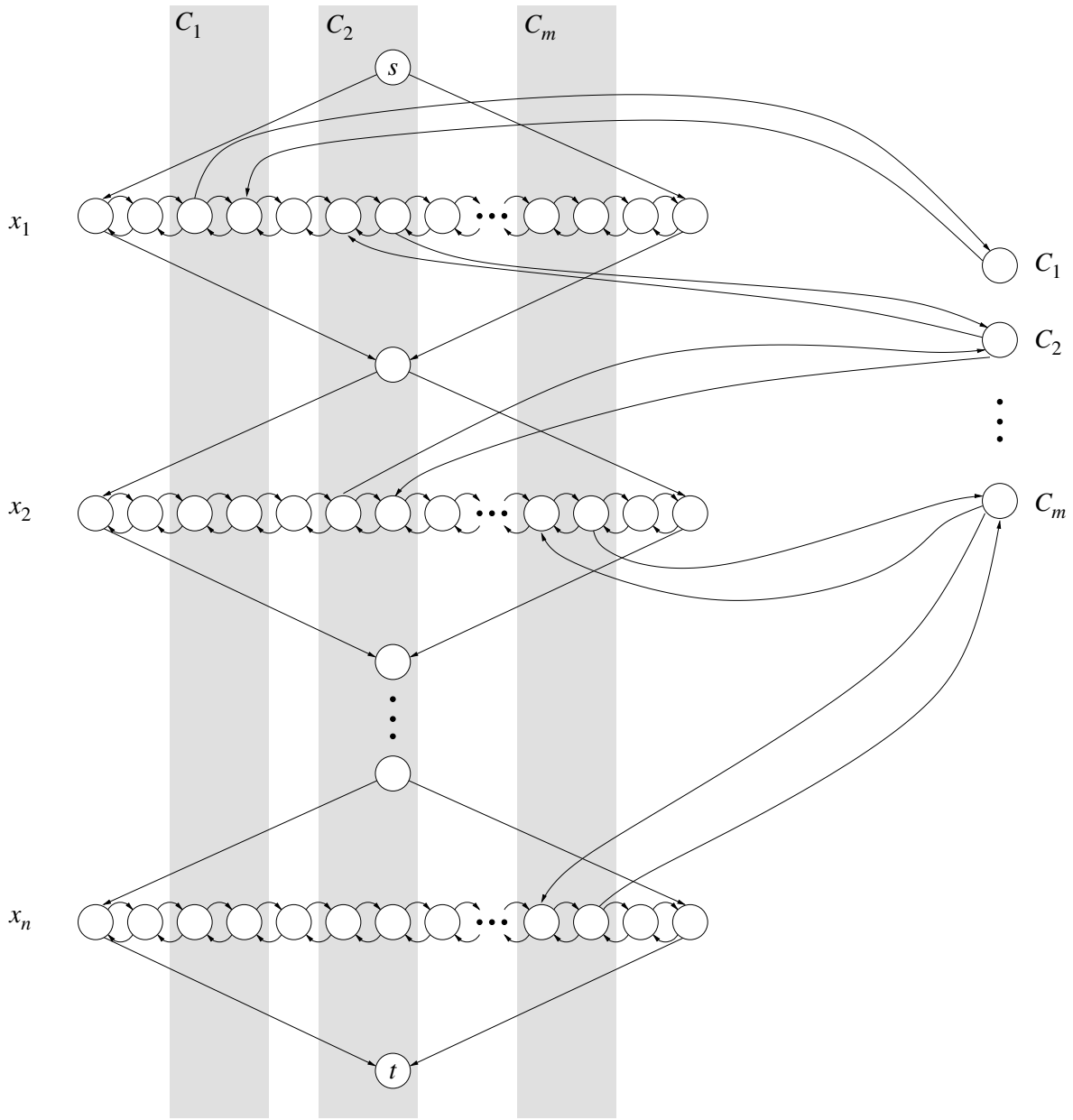



Figure 3.11: Representation of formula $(x_1 \vee \cdots) \wedge (\neg x_1 \vee x_2 \vee \cdots) \wedge \cdots \wedge (\neg x_2 \vee \cdots \neg x_n)$.

With these provisions, a clause node $C_i$ can be visited in a Hamiltonian path if and only if it is connected to a variable in a way that is compatible with its sense of traversal.

If a truth assignment satisfies a clauses, then the corresponding path through the diamond chain can be extended to visit the corresponding clause node; on the other hand, if an assignment does not satisfy a clause, there is no way to include the clause's node in the path without skipping or revisiting some other nodes, so that the path isn't Hamiltonian anymore. As an example, consider the truth assignment $x_1 = x_n = \top$, $x_2 = \bot$ and Fig. 3.12. Then, no detour can be made to visit node $C_2$ while keeping the path Hamiltonian. On the other hand, clauses $C_1$ and $C_m$ can be visited by "cutting" the chain in the appropriate places.

Therefore, a Hamiltonian path exists in the constructed graph if and only if the equation is satisfiable.s!on graphs □

### 3.8.2 Directed Hamiltonian cycles

We can remove the two special nodes $s$ and $t$ by requiring the path to be a cycle:

**Definition 36.** *Given a directed graph $G = (V, E)$ a Hamiltonian cycle in $G$ is a closed path (i.e., the initial node is also the final one) where every node in $V$ is visited exactly once (clearly, the first and last step, starting and ending at the same node, count as one visit).*

The corresponding problem can be stated as

**Definition 37** (DIRECTED HAMILTONIAN CYCLE)**.** *Given a directed graph $G = (V, E)$, does $G$ have a Hamiltonian cycle?*

The reduction discussed above still works just by adding an edge from $t$ to $s$. Any Hamiltonian cycle must contain that edge, because it is the only way to navigate back once the diamonds have been traversed. Therefore:

**Theorem 35.** *DIRECTED HAMILTONIAN CYCLE is **NP**-complete.*

### 3.8.3 Undirected Hamiltonian cycles

The reduction above is strictly dependent on the direction of edges to enforce detours only in precise conditions. However, we can easily reduce DIRECTED HAMILTONIAN CYCLE to its undirected version:

**Definition 38** (HAMILTONIAN CYCLE)**.** *Given an undirected graph $G = (V, E)$, does $G$ have a Hamiltonian cycle?*

The problem is somewhat less constrained, since an edge can be traversed in two directions, which might make the problem harder or worse. Anyway, we can easily reduce DIRECTED HAMILTONIAN PATH to it by splitting every node $x$ of the directed graph in three nodes, called $x_{\text{in}}$, $x_{\text{m}}$ and $x_{\text{out}}$ (see Fig. 3.13) and connect them as follows:

- connect $x_{\text{in}}$ to $x_{\text{m}}$, and $x_{\text{m}}$ to $x_{\text{out}}$;
- for every edge $x \mapsto y$ in the directed graph, connect $x_{\text{out}}$ to $y_{\text{in}}$.

Fig. 3.14 shows an example of such reduction.

It is easy to see that if the directed graph has a Hamiltonian cycle, so does the undirected graph: every sequence of edges $x \mapsto y \mapsto z$ that traverses $y$ in the directed graph corresponds to the sequence $x_{\text{out}} - y_{\text{in}} - y_{\text{m}} - y_{\text{out}} - z_{\text{in}}$ that traverses all three nodes corresponding to $y$; conversely, every path in the undirected graph that traverses $y_{\text{in}}$ must proceed to $y_{\text{m}}$ and $y_{\text{out}}$ before exiting to other nodes, otherwise $y_{\text{m}}$ could be left out if no node can be visited twice.

Therefore:

**Theorem 36.** *HAMILTONIAN CYCLE is **NP**-complete.*

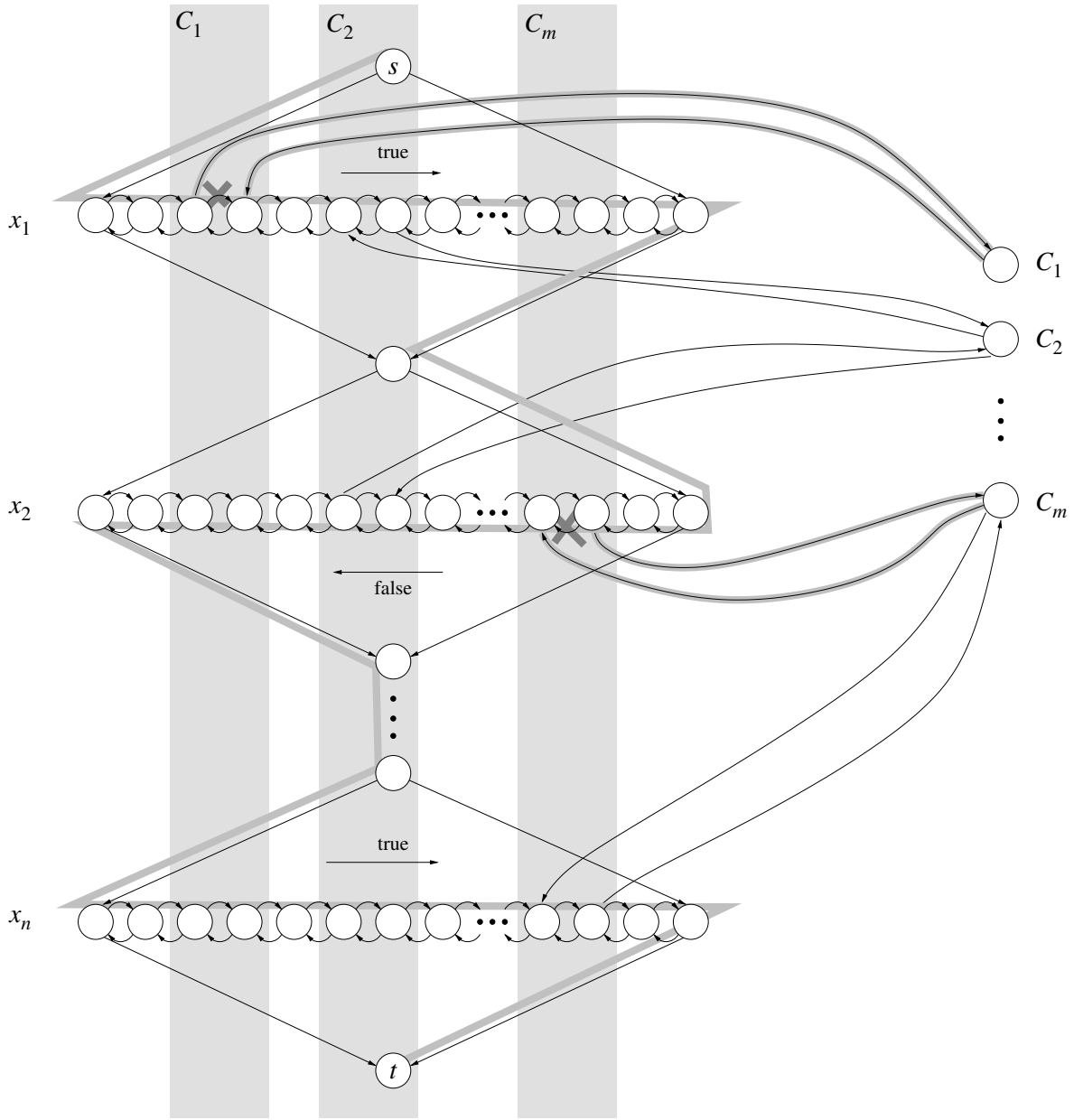



Figure 3.12:

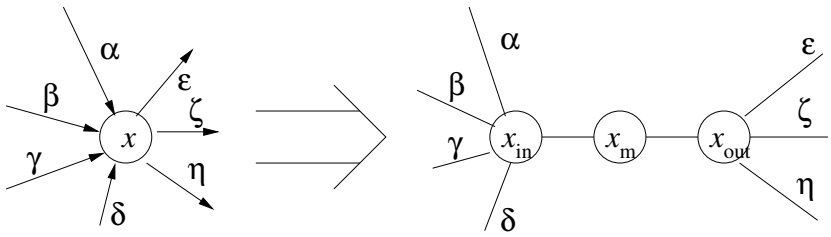



Figure 3.13: Splitting a node in a directed graph into three nodes in the undirected graph.

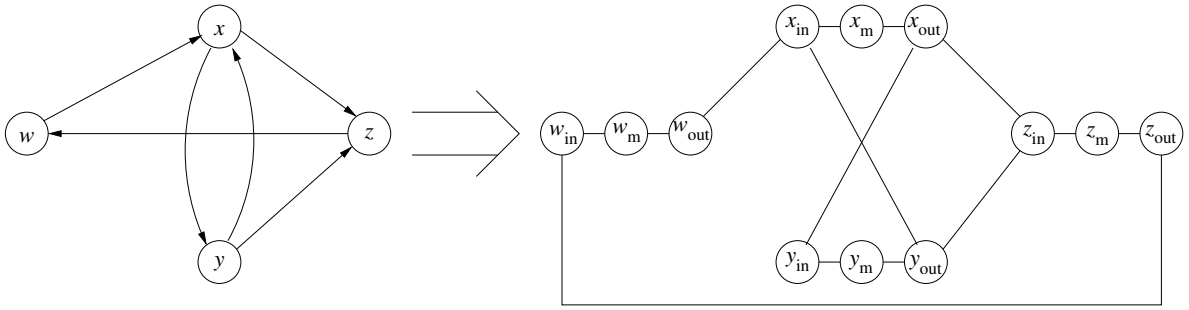



Figure 3.14: Converting a directed graph into an equivalent undirected graph.

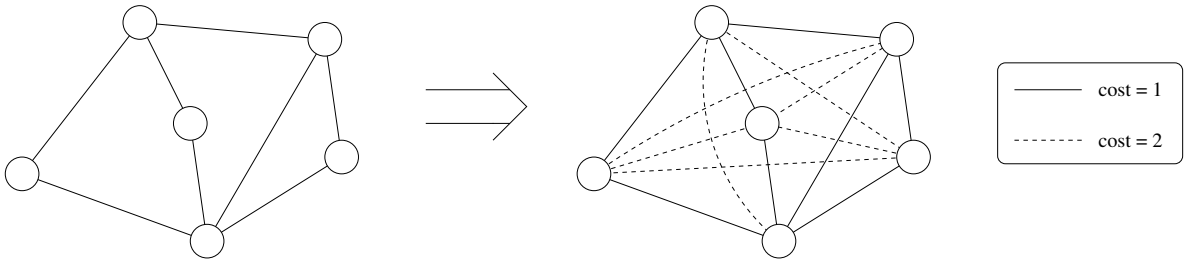



Figure 3.15: Reducing a generic HAMILTONIAN CYCLE instance to an equivalent TSP instance.

### 3.8.4 The Traveling Salesman Problem

We can reformulate the TSP in terms of Hamiltonian cycles:

**Definition 39** (Traveling Salesman Problem — TSP)**.** *Given a complete, undirected graph $G = (V, E)$, with numeric costs associated to edges ($c : E \to \mathbb{N}$) and a budget $k$, is there a Hamiltonian cycle in $G$ with overall cost not greater than $k$?*

**Theorem 37.** *TSP is **NP**-complete.*

*Proof.* We already know that TSP $\in$ **NP**.

To prove completeness, let us reduce HAMILTONIAN CYCLE to TSP. Given the undirected graph $G = (V, E)$, let us assign cost 1 to all its edges, then complete it adding all missing edges with cost 2. Clearly, the original graph $G$ has a Hamiltonian cycle if and only if the complete version has a Hamiltonian cycle with cost $|V|$ (i.e., composed of $|V|$ edges with cost 1). □

As an illustration, consider Fig. 3.15: the 6-node left-hand side graph has a Hamiltonian cycle if and only if the right-hand side graph has a TSP solution of cost 7 (i.e., visiting all 7 vertices once and not traversing any dashed edge with cost 2).

### 3.8.5 Relationship between P, NP and coNP

Clearly, $\mathbf{P} \subset \mathbf{NP} \cap \mathbf{coNP}$ because in **P** positive and negative answers are both polynomially verifiable. Currently, we don't know if the inclusion is strict or not.

Consider the decision version of the well-known integer factorization problem:

**FACTORING** — Given two positive integers $n, k \in \mathbb{N}$ does $n$ have a prime factor $p \geq k$?

In other words, in order to have a decision problem we do not ask for the factor itself, we just compare it with a target value.

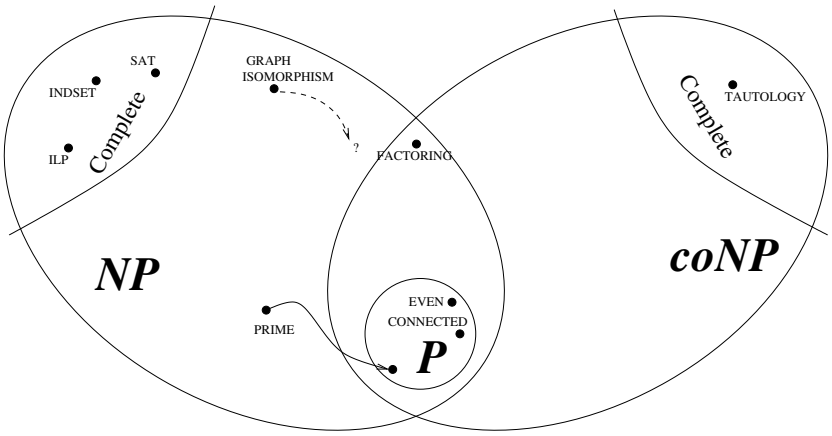



Figure 3.16: What we know up to now. If any of the **NP** or **coNP**-complete problems were to be proven in **P**, then all sets would collapse into it.

We do not know of any polynomial algorithm for integer factorization. Most numbers have small prime factors, and are easily decomposable, but some (namely, products of two large primes) are hard. Therefore, we cannot prove (yet) that FACTORING $\in \mathbf{P}$. However:

**Theorem 38.** *FACTORING* $\in \boldsymbol{NP} \cap \boldsymbol{coNP}$.

*Proof.* Suppose we are given the input instance $n, k \in \mathbb{N}$. Observe that, if we assume the usual positional representation for integers (e.g., base-2 or base-10 representation), then the input size is $O(\log n + \log k)$; moreover, since the question makes sense only if $k < n$, then the input size is $O(\log n)$.

- If the answer is yes, meaning that there is a prime number $p \geq k$ that divides $n$, such prime is an acceptable polynomial certificate. To verify it:
  1. the size of $p$ is smaller than the size of $n$ (therefore, the certificate is polynomially-sized wrt the instance);
  2. check that $p$ is actually prime (there is a polynomial-time algorithm for that),
  3. check that $p \geq k$,
  4. check that $p$ divides $n$.
- If the answer is no, meaning that all prime factors of $n$ are smaller than $k$, the list of all $m$ prime factors $p_1, p_2, \ldots, p_m$ of $n$ is a polynomial certificate for the negative answer. In fact, we can verify it as follows:
  1. the list is composed of $m \leq \log_2 n$ numbers, each requiring $O(logn)$ symbols for its representation; the size of the list is therefore $O((\log n)^2)$, polynomial wrt the input size;
  2. check that all numbers in the list $p_1, p_2, \ldots, p_m$ are prime (again, we use $m \leq \log_2 n$ applications of the polynomial-time primality algorithm);
  3. check that all $p_1, p_2, \ldots, p_m$ are less than $k$;
  4. check that $p_1 p_2 \ldots p_m = n$.

□

Fig. 3.16 summarizes what has been said in this chapter.

# Chapter 4

# Other complexity classes

Not all languages are **NP** or **coNP**. It is possible to define languages with higher and higher complexity.

## 4.1 The exponential time classes

It is possible to define classes that are analog to **P** and **NP** for exponential, rather than polynomial, time bounds:

**Definition 40.**

$$\boldsymbol{EXP} = \bigcup_{c=1}^{\infty} DTIME\left(2^{n^c}\right), \qquad \boldsymbol{NEXP} = \bigcup_{c=1}^{\infty} NTIME\left(2^{n^c}\right),$$

*and, of course,*

$$\boldsymbol{coNEXP} = \{L \subseteq \Sigma^* : \bar{L} \in \boldsymbol{NEXP}\}.$$

In short, **EXP** is the set of languages that are decidable by a deterministic Turing machine in exponential time (where "exponential" means a polynomial power of a constant, e.g., 2); **NEXP** is the same, but decidable by a NDTM. In other words, a language $L$ is in **NEXP** when $x \in L$ iff there is an exponential-sized (wrt $x$) certificate verifiable in exponential time. Finally, **coNEXP** is the set of exponentially-disprovable languages.

Coming up with languages that are in these classes, but not in **NP** or **coNP** is harder. One "natural" language is the equivalence of two regular expressions under specific limitations to their structure.

The following result should be immediate:

**Lemma 3.**

$$\boldsymbol{P} \subseteq \boldsymbol{NP} \subseteq \boldsymbol{EXP} \subseteq \boldsymbol{NEXP}.$$

*Proof.* The only non trivial inclusion should be **NP** $\subseteq$ **EXP**, but we just need to note that a non-deterministic machine with polynomial time bound can clearly be simulated by a deterministic machine in exponential time by performing all computations one after the other. □

Fig. 4.1 summarizes the addition of the exponential classes.

### 4.1.1 The "Restricted" Halting Problem

We already know (since Theorem 6) that the problem whether a deterministic TM $\mathcal{M}$ will halt on a given input $x$ is in general undecidable. However, we also know that, due to the existence of universal

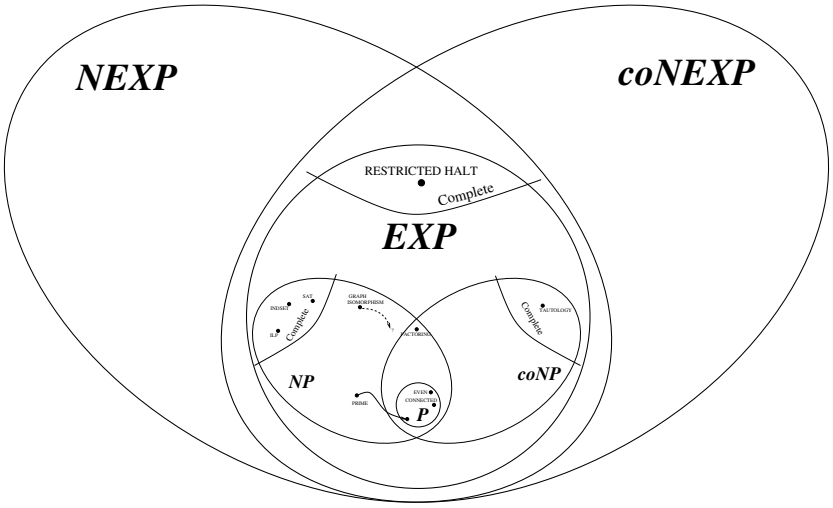



Figure 4.1: The exponential classes. The inner part is shown in greater detail in Fig. 3.16.

TMs, we are able to simulate any computation $\mathcal{M}(x)$ for an arbitrary number of steps. Thus, the following language is computable:

$$\text{RESTRICTED HALT} = \left\{ (\mathcal{M}, x, t) \quad : \quad \begin{array}{c} \mathcal{M} \text{ is a DTM} \wedge x \text{ is a string in } \mathcal{M}\text{'s alphabet} \\ \wedge\, T \in \mathbb{N} \wedge T_{\mathcal{M}}(x) \leq t \end{array} \right\}.$$

In other words, RESTRICTED HALT is the set of all TMs that halt on a given input within a given number of steps.

Then we can easily verify the following:

**Theorem 39.**

$$\textit{RESTRICTED HALT} \in \boldsymbol{EXP}.$$

*Proof.* In order to check whether the triplet $(\mathcal{M}, x, t)$ belongs to RESTRICTED HALT, we can emulate the computation of $\mathcal{M}(x)$ for at most $t$ steps (less, if it halts sooner) using a universal TM $\mathcal{U}$.

Let $m$ be the size of $\mathcal{M}$'s representation in $\mathcal{U}$, $n = |x|$ the size of the input string, and $s = O(\log t)$ the size of the representation of the number of steps, then the triplet $(\mathcal{M}, x, t)$ is represented as an input string of $O(m + n + s)$ symbols.

Since the emulation must be carried on for at most $t = 2^{O(s)}$ steps, each requiring possibly some scans of $\mathcal{M}$'s representation, then the whole simulation will take time $O(m \cdot 2^{ks})$ for some constant $k$, therefore requiring exponential time with respect to the input size. □

In order to better understand this proof, note that the number of steps that we need to emulate is, of course, linear with respect to the *value* represented by the input ($t$), but it is *exponential* with respect to the number $s$ of symbols required to represent $t$ on the tape.

It is very unlikely that RESTRICTED HALT $\in$ **NP**: for that, we would need a polynomial-size certificate that allows us to "skip" the exponential number $t$ of simulated steps required to prove that $\mathcal{M}(x)$ halts within time $t$.

In fact, we can see that RESTRICTED HALT is "the hardest" language in its class:

**Theorem 40.** *RESTRICTED HALT is **EXP**-complete (with respect to polynomial-time reductions).*

*Sketch of the proof.* We need to prove prove that every language in **EXP** has a polynomial-time reduction to RESTRICTED HALT.
Let $L \in$ **EXP** be one such generic language, and let $\mathcal{M}_L$ be an exponential-time TM that decides $L$.

In particular, let $p$ be a polynomial such that $\mathcal{M}_L(x)$ halts within $2^{p(|x|)}$ steps for every input string $x$. We can tweak $\mathcal{M}_L$ into a new machine $\mathcal{M}'_L$ that, instead of rejecting, runs forever (we turn a machine that "decides" $L$ into a new one that merely "accepts" it):

$\mathcal{M}'_L$ **on input** $x$:
**if** $\mathcal{M}(x)$ **accepts**
  **then accept**
  **else** run forever

Therefore, we have created a machine $\mathcal{M}'_L$ that halts within $t = 2^{p(|x|)}$ steps if and only if the original machine halted in an accepting state, otherwise won't halt within that time limit:

$$x \in L \qquad \Leftrightarrow \qquad (\mathcal{M}'_L, x, 2^{p(|x|)}) \in \text{RESTRICTED HALT.}$$

Since the reduction $x \mapsto (\mathcal{M}'_L, x, 2^{p(|x|)})$ can be computed in polynomial time with respect to $|x|$, we have

$$L \leq_{\text{P}} \text{RESTRICTED HALT.}$$

□

## 4.2 Space complexity classes

Up to this point, we considered time (expressed as the number of TM transitions) as the only valuable resource. Still, one may envision cases in which space constraints are more important. In order to provide a significant definition of space, we need to just consider *additional* space with respect to the input. In this Section we will use Turing machines with at least two tapes, the first one being a read-only tape containing the input string, which won't count towards space occupation.

**Definition 41.** *Given a computable function $f : \mathbb{N} \to \mathbb{N}$, $DSPACE\big(f(n)\big)$ is the class of languages $L$ that are decidable in space bounded by $O\big(f(|x|)\big)$, where $n$ is the size of the input; i.e., $L \in DSPACE\big(f(n)\big)$ if there is a multi-tape TM $\mathcal{M}$, with a read-only input tape, such that $\mathcal{M}$ decides $x \in L$ by using $O\big(f(|x|)\big)$ cells in the read/write tape(s).*

Note that, since we exclude the input tape from the computation, we allow for space complexities that are less than linear, such as DSPACE(1) or DSPACE($\log n$). This contrasts with time complexity classes which assume at least linear time because of the time needed to read the input.

We can introduce the equivalent non-deterministic class:

**Definition 42.** *Given a computable function $f : \mathbb{N} \to \mathbb{N}$, $L \in NSPACE\big(f(n)\big)$ if there is a multi-tape non-deterministic TM $\mathcal{N}$, with a read-only input tape, such that $\mathcal{N}$ decides $x \in L$ by using $O\big(f(|x|)\big)$ cells in the read/write tape(s).*

### 4.2.1 Logarithmic space classes: L and NL

**Definition 43.**

$$\boldsymbol{L} = DSPACE(\log n)$$

*is the class of languages that are decidable by a deterministic TM using logarithmic read-write space;*

$$\boldsymbol{NL} = NSPACE(\log n)$$

*is the same if non-deterministic computations are allowed.*

```
on input G = (V, E); s, t ∈ V
  current ← s  "current" is a counter on the working tape
  repeat |V| times
    if current = t
      then accept and halt
    non deterministically here the computation splits
      current ← a node adjacent to current among all adjacent nodes
  reject and halt
```

Figure 4.2: A non-deterministic, logarithmic-space algorithm for the ST-CONNECTIVITY language.

Note that if the input encodes a data structure, such as a graph or a Boolean formula, then a counter or a pointer referring to it has size $O(\log n)$ (in order to write numbers up to $n$ we need $O(\log n)$ symbols), therefore **L** contains all languages decidable by a constant number of pointers/counters.

Observe that if space is bounded by $c \log n$, then the machine can have at most $O(2^{c \log n}) = O(n^c)$ configurations, and therefore it must halt within that number of steps. Therefore

**Theorem 41.**

$$\boldsymbol{L} \subseteq \boldsymbol{P}, \qquad \boldsymbol{NL} \subseteq \boldsymbol{NP}.$$

**Examples**

The language

$$\text{POWER OF TWO} = \{1^{2^i} : i \in \mathbb{N}\}$$

of sequences of ones whose length is a power of two is in **L**. In fact, in order to determine the length of a string we just need a counter, whose size if logarithmic with respect to the input string.

**Definition 44.** *A triplet composed of a* directed *graph $G = (V, E)$ and two nodes $s, t \in V$ belongs to the ST-CONNECTIVITY (or STCON, or REACHABILITY) language if there is a path in $G$ from $s$ to $t$.*

Note that the definition is about a directed graph, and it requires a path from a specified source node $s$ to a specified target node $t$.

Observe that a non-deterministic TM can simply keep in its working tape a "current" node (initially $s$), and non-deterministically jump from the current node to any connected node following the graph's adjacency matrix, as shown in Fig. 4.2.

If there is a path from $s$ to $t$, then one of the computations will be lucky enough to follow it and terminate in an accepting state within $|V|$ computations; otherwise, no computation will be able to reach $t$ and all will terminate in a rejecting state after $|V|$ iterations. Note that an actual NDTM implementation will require space for a current node, an iteration counter and possibly some auxiliary variables which all need to contain numbers from 1 to $|V|$. Therefore, the amount of space needed is bounded by $c \cdot \log |V|$. Since the input must contain an adjacency matrix, which is quadratic with respect to $|V|$, its size is $|x| = O(|V|^2)$. Therefore,

**Theorem 42.**

$$\mathit{STCON} \in \boldsymbol{NL}.$$

Any known efficient deterministic algorithm for STCON requires linear space (we have to maintain a queue of nodes, or at least be able to mark nodes as visited). While we don't conclusively know if STCON also belongs to **L**, we can prove the following (this result, due to Walter Savitch, is the basis of Savitch's Theorem 46):

**Theorem 43.**

$$\mathit{STCON} \in \mathit{DSPACE}\big((\log n)^2\big).$$

*Proof.* The following algorithm only requires $(\log n)^2$ space, even though it is extremely inefficient in terms of time:

```
on input G = (V, E); s, t ∈ V
  path_exists ← function (v, w, l)
    if l = 0
      return false                                   No more steps allowed.
    if l = 1                                         If only one step remains,
      return (v, w) ∈ E                              either v points directly to w, or nothing.
    for all v' ∈ V
      if path_exists(v,v',⌊l/2⌋) ∧ path_exists(v',w,⌈l/2⌉)
        return true
    return false
  if path_exists (s, t, |V|)
    accept and halt
  else
    reject and halt
```

The function `path_exists` tells us if there is a path from the generic node $v \in V$ to the generic node $w \in V$ having length at most $l$. It is recursive: in the base cases, it tests if $v$ and $w$ are directly connected or are the same node (in which case the path obviously exists). Otherwise, the following property is true: *if a path of length $l$ exists from $v$ to $w$, then we can find a node $v'$ in the middle of it*, in the sense that the paths from $v$ to $v'$ and from $v'$ to $w$ have length $l/2$ (give or take one if $l$ is odd). The function searches for this middle node $v'$ by iterating through all nodes in $V$; this is extremely inefficient in terms of time, but it allows the application of a *divide et impera* strategy that keeps the recursion depth to $\log l$.

Since the function requires only a constant number of variables to work, each of size $O(\log |V|)$, and that the call depth, starting from $l = |V|$, is again $O(\log |V|)$, remembering that the input size is $n = O(|V|^2)$, the conclusion follows. □

Observe that the proof outline doesn't explicitly define a TM; however, a recursive call stack can be stored in a TM tape as contiguous blocks of cells.

Another way to understand the algorithm in the proof above is the following: if we replaced the recursive calls with the following pair, we would obtain the usual step-by-step path search (albeit still rather inefficient):

$$\texttt{path_exists}(v,v',1) \wedge \texttt{path_exists}(v',w,l-1).$$

While Savitch's solution requires a large number of steps, but is able to keep the recursive depth logarithmic with respect to the problem size, the step-by-step alternative takes much fewer steps, but the recursion depth is linear[1].

### 4.2.2 NL-completeness of STCON

In the proof of the Cook-Levin theorem (see Fig. 3.7, page 49) we characterized the computation of a NDTM as a directed acyclic graph whose nodes are *configurations* linked by steps, each consisting of the application of possibly more than one transition rule.

**Theorem 44.** *STCON is **NL**-complete.*

*Proof.* We already know that STCON $\in$ **NL**.

Given a NDTM $\mathcal{N}$, with a read-only input tape and a read/write tape (i.e., of the type considered for logspace-constrained languages), and an input string $x$, the computation can be represented as a directed acyclic graph of configurations, where every configuration is composed of the following items:

[1] See `https://comp3.eu/savitch.py` for a small python script that lets you compare the two approaches

- the contents of the working tape, initially empty;
- the position on the input tape;
- the position on the working tape;
- the state, initially $q_0$.

If $\mathcal{N}$ is logarithmic-space constrained, note that:

- the content $x$ of the input tape needn't be part of the configuration description, because it never changes; only the current position on that tape needs to be recorded. As such, the description of a configuration occupies $O(\log |x|)$ space;
- we know that every branch of the computation eventually terminates, therefore the resulting graph does not contain loops: it is a "directed acyclic graph" (DAG);
- every branch of the computation (every path in the graph starting from the initial configuration) proceeds until it reaches a halting state, corresponding to two possible configurations, the *accepting* and the *rejecting* one[2].

From the above discussion, we can reduce every language $L \in \mathbf{NL}$ to STCON by the following steps:

- Let $L \in \mathbf{NL}$; therefore, there is a NDTM $\mathcal{N}_L$ that decides $x \in L$ by using no more than $O(\log |x|)$ cells in the working tape (and having $x$ stored in the read-only input tape as usual).
- We know that the question "$x \in L$?" is equivalent to "does the computation $\mathcal{N}_l(x)$ have at least one accepting branch?"
- The latter can be expressed in terms of STCON: "In the computation graph of $\mathcal{N}(x)$, is there a path from the initial configuration node to the node ACCEPT?"
- As all nodes have a $O(\log |x|)$-sized description, as pointed out above, the non-deterministic algorithm for STCON from Fig. 4.2 can actually answer this question in $O(\log |x|)$ space.

The described reduction itself takes logarithmic space, because to run STCON we just need to know if two arbitrary configurations (nodes in the graph) are connected or not, and this can be done directly from a description of $\mathcal{N}_L$, without requiring additional space to actually store the graph. □

We can sum up what we have seen up to now in the following theorem:

**Theorem 45.** ***NL*** $\subseteq$ ***P***.

*Proof.* Let $L \in \mathbf{NL}$; since STCON is **NL**-complete, there is a logarithmic-space (and therefore polynomial-time) reduction $f$ mapping instances $x$ of $L$ into instances $f(x)$ of STCON. Let $\mathcal{M}_{\text{STCON}}$ be a deterministic TM that decides STCON; therefore, the concatenation $\mathcal{M}_{\text{STCON}}(f(x))$ decides $x \in L$ in polynomial time, thus $L \in \mathbf{P}$. □

### 4.2.3 Polynomial space: PSPACE and NPSPACE

As we did for **P** and **NP**,

**Definition 45.**

$$\begin{aligned} \textbf{\textit{PSPACE}} &= \bigcup_{c=0}^{\infty} \mathit{DSPACE}(n^c), \\ \textbf{\textit{NPSPACE}} &= \bigcup_{c=0}^{\infty} \mathit{NSPACE}(n^c). \end{aligned}$$

[2]There might be many different accepting configurations, and many rejecting ones; however, we can always consider all "ACCEPT" nodes in the graph as being the same node; same for "REJECT" nodes.

The following inequalities should be obvious enough: **PSPACE** $\subseteq$ **NPSPACE** (as determinism is a special case of nondeterminism), **P** $\subseteq$ **PSPACE** (as having polynomial time allows us to touch at most a polynomial chunk of tape), **NP** $\subseteq$ **NPSPACE** (same reason).

A very important result shows that nondeterminism is less important for space-bounded computations: renouncing nondeterminism causes at most a quadratic loss.

**Theorem 46** (Savitch's theorem)**.** *Given a function $f(n)$,*

$$NSPACE\big(f(n)\big) \subseteq DSPACE\big(f(n)^2\big).$$

*Proof.* Consider a language $L \in \text{NSPACE}\big(f(n)\big)$ and a generic input $x$. Then there is a NDTM $\mathcal{N}$ that decides $x \in L$ by using at most $O\big(f(|x|)\big)$ tape cells. The number of different configurations of the machine is therefore bounded by $N_c = 2^{O(f(|x|))}$. As in the proof of Theorem 44, let us consider the directed graph $G = (V, E)$ having all possible $N_c$ configurations as the set $V$ of nodes, and with an edge $(c_1, c_2) \in E$ if there is a transition rule in the $NDTM$ that allows transition from $c_1$ to $c_2$. Every path in $G$ represents a possible computation of the machine, starting from an arbitrary configuration.

Let us call $s \in V$ the initial configuration of $\mathcal{N}$ with input $x$. Obviously, $x \in L$ if and only if there is an accepting computation of $\mathcal{N}$ with input $x$, i.e., if and only if there is a path in $G$ from $s$ to an accepting configuration. Let us add a new node $t$ to $V$, and an edge from every accepting state to $t$. At this point, $x \in L$ if and only if there is a path from $s$ to $t$ in $G$, therefore if and only if $(G, s, t) \in \text{STCON}$.

From Theorem 43, this STCON problem can be decided in space $O\big((\log N_c)^2\big) = O\Big(\big(\log 2^{O(f(|x|))}\big)^2\Big) = O\big(f(|x|)^2\big)$. □

This is an immediate consequence:

**Corollary 9.**

$$\boldsymbol{PSPACE = NPSPACE}.$$

*Proof.*

$$\textbf{PSPACE} \subseteq \textbf{NPSPACE} = \bigcup_{c=0}^{\infty} \text{NSPACE}(n^c) \subseteq \bigcup_{c=0}^{\infty} \text{DSPACE}(n^{2c}) = \textbf{PSPACE}.$$

□

At last, getting all previous results together, we obtain the following chain of inclusions:

$$\textbf{L} \subseteq \textbf{NL} \subseteq \textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE} = \textbf{NPSPACE} \subseteq \textbf{EXP} \subseteq \textbf{NEXP}.$$

To summarize our findings:

- **L** $\subseteq$ **NL** — because deterministic machines are a special case of non-deterministic ones;
- **NL** $\subseteq$ **P** — because STCON is **NL**-complete and is decidable in deterministic polynomial time;
- **P** $\subseteq$ **NP** — because deterministic machines are a special case of non-deterministic ones;
- **NP** $\subseteq$ **NPSPACE** — because within a polynomial number of steps a TM (either deterministic or non-) can visit at most a polynomial number of cells;
- **PSPACE** = **NPSPACE** — because of Savitch's theorem, which implies that every non-deterministic polynomial space-bound machine can be simulated by a deterministic machine in polynomial space, albeit with a possibly quadratic penalty;
- **PSPACE** $\subseteq$ **EXP** — because a TM using $O(n^k)$ cells can have no more than $O(c^{n^k})$ (i.e., an exponential number of) configurations, and a machine that halts cannot visit the same configuration twice;
- **EXP** $\subseteq$ **NEXP** — again, because deterministic machines are a special case of non-deterministic ones.

### 4.2.4 The Quasi-polynomial time class

In Theorem 43 we showed that STCON $\in$ DSPACE$\big((\log n)^2\big)$. Let us give a name to this space class:

$$\mathbf{L}^2 = \text{DSPACE}\big((\log n)^2\big).$$

Informally, $\mathbf{L}^2$ is the class of languages that can be decided by using a logarithmic number of counters, pointers and flags (for comparison, **L** requires a *constant* number of counters ,pointers and flags).

Let us try to derive a time bound from the space constraint. If a deterministic machine uses $O\big((\log n)^2\big)$ tape cells, then it has at least $O(2^{(\log n)^2})$ different configurations that cannot be repeated. Therefore, its time bound is at least

$$O(2^{(\log n)^2}) = O(2^{\log n \cdot \log n}) = O\big((2^{\log n})^{\log n}\big) = O(n^{\log n})$$

(give or take a few constants). We can rewrite this as

$$\mathbf{L}^2 = \text{DSPACE}\big((\log n)^2\big) = \text{DTIME}(n^{\log n}).$$

Note that $n^{\log n}$ grows faster than any constant power $n^k$ (trivially, whichever value we choose for $k$, $\log n$ will sooner of later become larger).

In general, we define the *quasi-polynomial* deterministic time class as the class of languages that are decidable in a time bounded by some polylogarithmic power of n:

$$\mathbf{QP} = \bigcup_{k=0}^{\infty} \text{DTIME}(n^{(\log n)^k}).$$

Clearly, $\mathbf{P} \subseteq \mathbf{QP}$. In general, **QP** is an intermediate class between **P** and **NP**, and it is not expected to contain any **NP**-complete language. GRAPH ISOMORPHISM is an important problem in **QP**.

# Part II

# Additional material (not required)

# Chapter 5

# Topics from previous editions

This chapter contains material that was presented in previous editions of the course, but has been removed in later editions for reasons of time and clarity. These topics won't appear in the exams.

## 5.1 A set which we are (currently) unable to enumerate

In order to build some more intuition, let's start with a very simple problem that has been baffling mathematicians for centuries.

### 5.1.1 Collatz sequences

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*[1] 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?"

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.

### 5.1.2 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. Figure5.1 details two possible

[1] `https://en.wikipedia.org/wiki/Collatz_conjecture`

```
function collatz (n ∈ ℕ \ {0}) ∈ {0,1}
  repeat
    if n = 1 then return 1
    if n is even
      then n ← n/2
      else n ← 3n + 1
  return 0
```

```
function collatz (n ∈ ℕ \ {0}) ∈ {0,1}
  return 1
```

Figure 5.1: 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.

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[2]). 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**[3] (or RE, in short).

Having a procedure that only terminates when the answer is "yes" might not seem much, but it actually allows us to enumerate all numbers having the property. The function in Fig. 5.2 shows the basic trick to enumerate a potentially non-recursive set, applied to the Collatz sequence: the **diagonal method**[4]. 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.

## 5.2 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[5]:

**Definition 46** (Post Correspondence Problem — PCP)**.** *Given two sets of $n$ strings, $\{A_1, \ldots, A_n\} \subset \Sigma^*$ and $\{B_1, \ldots, B_n\} \subset \Sigma^*$, is it possible to find a finite sequence of $k$ indices $1 \leq i_1, \ldots, i_k \leq n$ (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 $\{(A_1, B_1), (A_2, B_2), \ldots, (A_n, B_n)\} \subset \Sigma^* \times \Sigma^*$ 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:

[2] To the best of my knowledge, which isn't much.

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

[4] See `https://comp3.eu/collatz.py` for a Python version.

[5] 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`

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

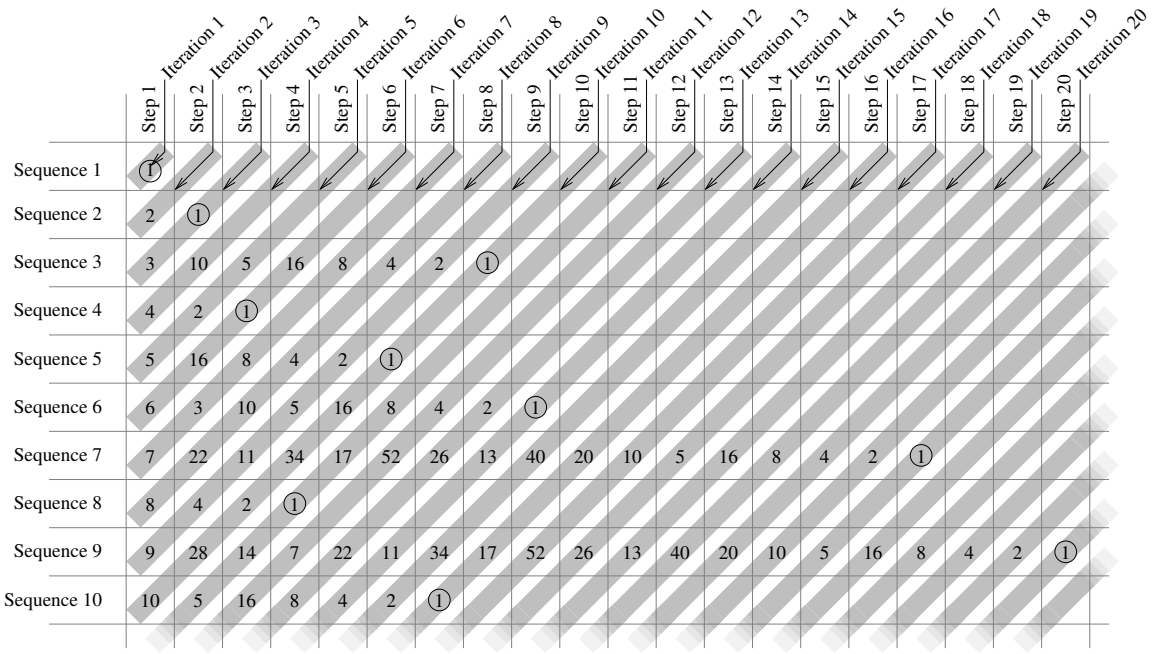
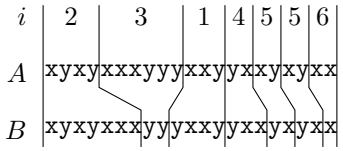



Figure 5.2: Enumerating all Collatz numbers: top: the algorithm; bottom: a working schematic

| $i$ | $A_i$ | $B_i$ |
|---|---|---|
| 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:

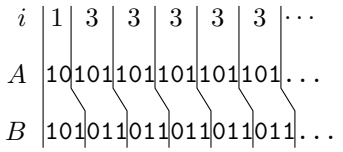



Note that this is actually the concatenation of two simpler solutions: $2, 3, 1$ and $4, 5, 5, 6$.

Some problems have no solution. For instance:

| $i$ | $A_i$ | $B_i$ |
|---|---|---|
| 1 | 10 | 101 |
| 2 | 011 | 11 |
| 3 | 101 | 011 |

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 $A$:




Let us consider another, simpler variant of the PCP:

**Definition 47** (Modified Post Correspondence Problem — MPCP)**.** *In the same conditions of PCP, we furthermore require that the first chosen index is $i_1 = 1$ (i.e., pair 1 is initially laid out).*

### 5.2.1 Undecidability of the Modified PCP

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

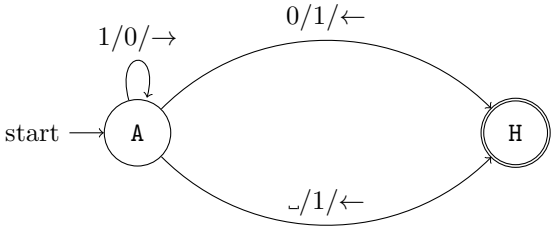
- $\mathcal{M}$ has a 3-symbol alphabet $\Sigma = \{␣, 0, 1\}$, where the default symbol is ␣;
- $\mathcal{M}$ never moves left of its starting position (i.e., the tape only extends indefinitely to the right);
- $\mathcal{M}$ never writes a ␣ (however it still has two symbols to write).

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.

In order to proceed from here, we need to recall the definition of computation of a TM as a sequence of configurations ad described in Section 3.5.2:

**Definition 48** (configuration)**.** *A "configuration" of a TM consists of three pieces of information:*

- *the content of the tape,*
- *the current position, and*
- *the current state.*

Therefore, a computation is a sequence of configurations, starting from the initial one and possibly ending with a halting state, as shown in Fig. 3.5.

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:

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

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.

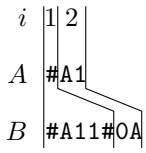
Suppose that in its initial configuration $\mathcal{M}$'s tape contains the string `11`, then its representation will be "`#A11#`", i.e., the tape's content with the state's symbol to the left of the current position, and delimiters`#` to enclose it. Since the first steps involves replacing the leftmost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

| $i$ | 1 |
|---|---|
| $A$ | # |
| $B$ | #A11# |

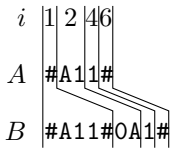
Note that the next character to match in $B$ is a state name, followed by a symbol. Since our transition rule requires the machine to replace the symbol and move right, whenever we find the string "`A1`" we know that the next configuration will need to contain "`0A`". That will be our second string pair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



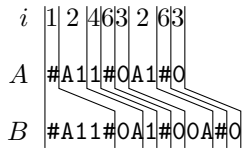

In order to complete the first step, all other symbols that are not in the current position must be copied, therefore we will need a bunch of other "copying" rules (one per tape symbol, one for the state separator)

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

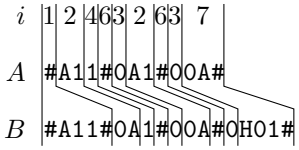
Note that these rules would make the original PCP trivial, but we are working with the modified version where an initial string is forced. With these new rules we can advance the matching:



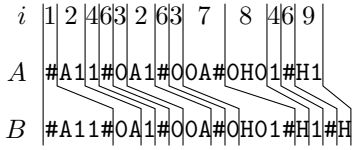

Note that the existing rules allow us to take the matching still further:



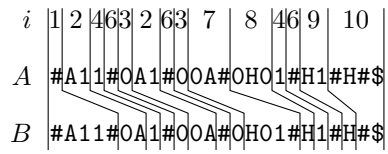

Note that in the configuration that we are currently trying to match the state letter is on the right of all tape symbols. This means that the current symbol is a blank, and $\mathcal{M}$'s transition rule requires to write "1", move left and halt. Since we need to move left, we cannot use the pair $i = 3$ to proceed, because the next symbol to appear in $B$ should be the state letter "H"; to move left, we introduce the pair $A_7 = \texttt{0A\#}$, $B_7 = \texttt{H01\#}$, 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 "$\sigma H$" or "$H\sigma$", where $\sigma$ 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 "\$": $A_{10} = \texttt{\#H\#\$}$, $B_{10} = \texttt{\#\$}$.




We have therefore constructed a Modified PCP instance that mimics 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 state set $Q$, 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 string $B$ 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 $\sigma = \textvisiblespace$, 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 $H$ 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 $A_i = \texttt{\#}H\texttt{\#\$}$, $B_i = \texttt{\#\$}$.
  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 47.** *The Modified Post Correspondence Problem is undecidable.*

### 5.2.2 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 $(A_1, B_1), (A_2, B_2), \ldots, (A_n, B_n)$ 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 original 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 $1, i_2, i_3, \ldots, i_k$ 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 $0, i_2, \ldots, i_k$ 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

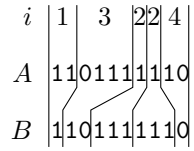
| $i$ | $A_i$ | $B_i$ |
|---|---|---|
| 1 | 11 | 1 |
| 2 | 1 | 111 |
| 3 | 0111 | 10 |
| 4 | 10 | 0 |

$\Rightarrow$

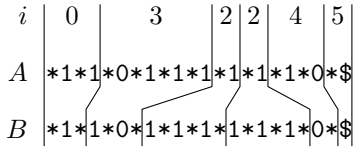
PCP

| $i$ | $A_i$ | $B_i$ |
|---|---|---|
| 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 48.** *The Post correspondence problem is uncomputable.*

## 5.3 Enhancing and restricting TMs

### 5.3.1 Oblivious Turing Machines

**Definition 49.** *A (one-tape) Turing Machine is said to be* oblivious *if the sequence of left-right moves it performs does not depend on the specific input string, but only on its length.*

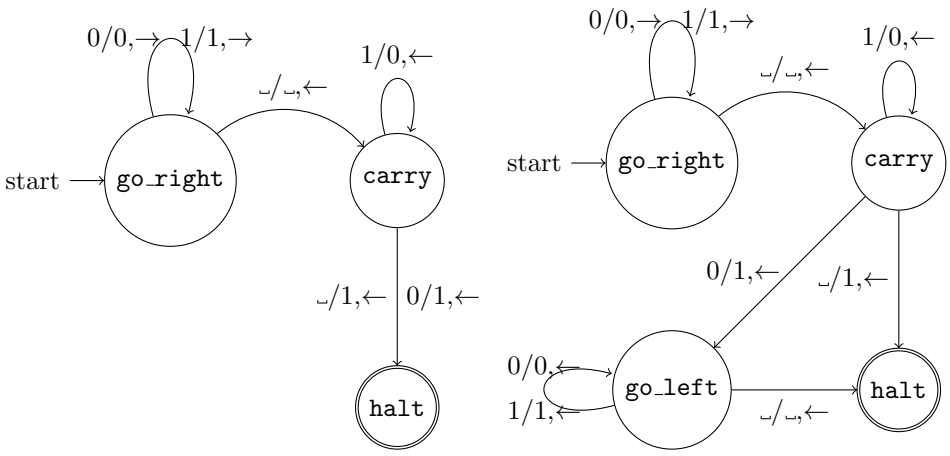



Figure 5.3: A non-oblivious TM (left) and an equivalent oblivious version (right).

The term is taken from computer security and cryptography: it refers to the inability for an observer who can only see the machine's movements, but not the actual tape contents, to learn anything about the machine's input — aside from its length.

As an example, consider the machine represented in the left-hand side of Fig. 5.3. It works on $\Sigma = \{\text{␣}, 0, 1\}$ and, assuming an input string $s \in \{0,1\}^*$ and start position in the leftmost non-blank symbol, increments the value represented by $s$ by 1. It scans the input left to right (state "go_right") until it finds a blank, then (state "carry") overwrites 1's with 0's until it finds the first non-1 symbol; then it writes a 1 and halts. The initial ("go_right") phase of the machine is oblivious: the motion does not depend on whether the symbol is 0 or 1, and it only ends when a blank is found (in this context, we interpret blanks as end-of-input markers). However, the carry phase ends as soon as a non-1 symbol is found, and therefore depends on the input symbols. An onlooker who only cares about the machine's moves learns something about the input string, namely how many 1's it ended with (i.e., the position of the rightmost 0).

To make the machine fully oblivious, it can simply keep moving left after the carry phase, leaving the rest of the input unchanged, and halting only when it finds the left blank delimiter. To do this (right-hand side of Fig. 5.3), we just add a state "go_left" that completes the left sweep before halting. At this point, for every input $s$ of size $n = |s|$, the machine performs precisely $n$ right moves, followed by $n + 2$ left moves before halting, regardless of the input symbols: an onlooker only learns the string length, but none of its bits.

This restriction does not impair the model's computational power: all computable functions can be computed by an oblivious machine (with a quadratic time loss).

**Theorem 49.** *Every 1-tape TM $\mathcal{M}$ with worst-case time $T_{\mathcal{M}}(n)$ on input size $n$ can be simulated by an oblivious TM $\mathcal{M}_{obl}$ with a quadratic increase on the worst-case time: $T_{\mathcal{M}_{obl}}(n) = O(T_{\mathcal{M}}(n)^2)$.*

*Proof.* Consider the proof of Theorem 3, concerning the emulation of a multiple-tape TM. The emulation technique presented in that proof is almost oblivious, as it requires a sequence of full input sweeps in order to emulate a single step of the $k$-tape machine.

Fundamentally, $\mathcal{M}_{\text{obl}}$ will use an extended alphabet with marked symbols to record the current position of $\mathcal{M}$ on the tape. Every step of $\mathcal{M}$ is emulated by sweeping the whole input left-to-right in order to acquire the symbol in the current position, then right-to-left to update the tape.

With a few more precautions, such as marking the beginning and end of the (potentially increasing) visited portion of the tape with special symbols, all tape sweeps can be made exactly similar to each other, with the exception of their length, which will increase by 1 at each end at every sweep.

The quadratic slowdown is due to the fact that we replace every step of $\mathcal{M}$ with a number of tape sweeps, and the size of the (visited portion of the) tape might grow by 1 at every step, and is therefore bounded by $O(T_{\mathcal{M}}(n))$. □

### 5.3.2 Allowing infinite states

The main reason why we restrict TMs to a finite number of states is that we want to define a practical computational model. Having an infinite number of states would be equivalent to having a computer with an infinitely long program. It is easy to prove that such assumption would make every function computable: we would be able to encode the complete input in the machine's state.

**Theorem 50.** *Every decision function is computable by a Turing Machine with a countable set of states.*

*Proof.* Let $\Sigma = \{\text{␣}, 0, 1\}$; let $f : \{0,1\}^* \to \{0,1\}$ be an arbitrary decision function on binary strings. We can define the TM that decides $f$ as follows.
Since $\Sigma^*$ is countable, let us index the states with strings: $Q = \{q_s | s \in \Sigma^*\}$, with $q_\varepsilon$ as initial state. Let the initial position be the leftmost symbol in the input.
Our machine will sweep the input string, erasing it and collecting the bit sequence in the state's index. As soon as the string is fully scanned, the machine reads a blank and replaces it with the value $f(s)$, leaving it as its final output before halting.
Here is the transition function that implements the machine:

$$\begin{array}{rcl} F : \Sigma \times Q & \to & \Sigma \times Q \times \{\leftarrow, \rightarrow\} \\ (\sigma, q_s) & \mapsto & \begin{cases} (\text{␣}, q_{s\sigma}, \rightarrow) & \text{if } \sigma \in \{0,1\} \\ (f(s), \text{HALT}, \rightarrow) & \text{if } \sigma = \text{␣} \end{cases} \end{array}$$

□

Two observations:

- Basically, the machine we just described implements a binary tree with unbounded depth, where every node is associated to the function's value for the corresponding sequence of edge labels. In other words, an infinite-state machine would allow us to build a program with an infinite chain of `if` statements.
- Note that we needn't be able to "build" the machine. What we can say is that for every possible decision function, even the Halting problem, there is an infinite-state TM that computes it, even though we might not be able to identify it.

In conclusion, machines with an infinite number of states are quite uninteresting for our purposes.

## 5.4 A relationship between exponential and polynomial time classes

We can show that the analysis of the relationship between **EXP** and **NEXP** can help wrt the **P** vs. **NP** problem. In particular,

**Theorem 51.** *If **EXP** ≠ **NEXP**, then **P** ≠ **NP**.*

*Proof.* We will prove the converse. Suppose that **P** = **NP**, and let $L \in$ **NEXP**. We shall build a deterministic TM that computes $L$ in exponential time.

Since $L \in$ **NEXP**, there is a NDTM $\mathcal{M}$ that decides $x \in L$ within time bound $2^{|x|^c}$.

We cannot hope to reduce an exponential computation to polynomial time. However, we can exponentially enlarge the input. Consider the language

$$L' = \left\{ \left( x, 1^{2^{|x|^c}} \right) : x \in L \right\}.$$

The language $L'$ is obtained from $L$ by padding all of its strings with an exponentially-sized string of 1's. Now, consider the following NDTM $\mathcal{M}'$ that decides$y \in L'$:

- Check whether $y$ is in the form $\left(x, 1^{2^{|x|^c}}\right)$ for some $x$ (not necessarily in $L$); if not, REJECT because $y \notin L'$;
- Clean the padding 1's, leaving only $x$ on the tape;
- Execute $\mathcal{M}(x)$ and ACCEPT or REJECT accordingly.

Now, each of the three outlined phases of $\mathcal{M}'$ have an exponential execution time wrt $x$, but a polynomial time wrt the much larger padded input $y$. Therefore, $L' \in \mathbf{NP}$.

Since we assumed $\mathbf{P} = \mathbf{NP}$, then $L' \in \mathbf{P}$, therefore there is a deterministic TM $\mathcal{N}'$ that decides $L'$ in polynomial time (wrt the padded size of strings in $L'$, of course).

But then we can define the deterministic TM that, on input $x$, pads it with $2^{|x|^c}$ ones (in exponential time), then runs $\mathcal{N}'$ on the resulting padded string. This machine is deterministic and accepts $L$ in exponential time, therefore $L \in \mathbf{EXP}$. □

## 5.5 The Merkle-Hellman cryptosystem

As a "real-world" application of the **NP**-completeness of SUBSET SUM, let us consider the following cryptosystem. Now broken, it was one of the earliest public-key cryptosystems[6] together with RSA.

**Description** Alice generates a sequence of $n$ integers $y_1, \ldots, y_n$ which is *super-increasing*, i.e., every item is larger that the sum of all previous ones:

$$y_i > \sum_{j=1}^{i-1} y_j qquad \text{for } i = 2, \ldots, n.$$

Notice that, given a super-increasing sequence, there is a simple algorithm to solve the SUBSET SUM problem for a given sum $s$:

```
function SUPERINCREASING_SUBSET_SUM (y_1, ..., y_n, s)      y_1, ..., y_n is super-increasing
    I ← ∅
    for i ← n..1                                            scan items starting from the largest
        if y_i < s                                          every time an item can be subtracted
            s ← s - y_i                                     subtract it
            I ← I ∪ {i}                                     record its index
    if s = 0
        return I                                            subtracted items added up to s
    else
        reject                                              the sum was not achievable
```

In order to scramble up her numbers, Alice chooses a positive integer $m > \sum_i y_i$ and another integer $r > 0$ such that $r$ and $m$ are coprime, i.e., $\gcd(r, m) = 1$. Next, she multiplies all the elements in her super-increasing sequence by $r$, modulo $m$:

$$x_i \equiv y_i \cdot r \pmod{m}. \tag{5.1}$$

[6] See `https://en.wikipedia.org/wiki/Merkle-Hellman_knapsack_cryptosystem`

In other words, $x_i$ is the remainder of the division of $y_i r$ by $m$. She finally publishes the numbers $x_1, \ldots, x_n$ as her public key.

When Bob wants to send a message to Alice, he encodes it into an $n$-bit string $(b_1, \ldots, b_n)$. He computes the sum

$$s = \sum_{i=1}^{n} b_i x_i,$$

where the $x_i$'s are the ones published by Alice, and sends $s$ to Alice.

Notice that the $x_i$'s have a basically random distribution in $\{0, \ldots, m-1\}$. They are not super-increasing, and the SUBSET SUM problem cannot be solved by a simple algorithm.

Alice, however, can move $s$ back to the super-increasing sequence by "undoing" the scrambling operation (5.1). Since she knows $r$ and $m$, which are kept secret, she can compute the inverse of $r$ modulo $m$, i.e., the only number $r' \in \{1, \ldots, m-1\}$ such that

$$r \cdot r' \equiv 1 \pmod{m}.$$

The algorithm to compute $r'$ is an extension of Euclid's gcd algorithm and is polynomial in the sizes of $r$ and $m$[7].

Alice can compute $s' \equiv s \cdot r' \pmod{m}$, which yields the same subset defined by Bob's binary string within the super-increasing sequence:

$$s' \equiv sr' \equiv \left( \sum_{i=1}^{n} b_i x_i \right) r' \equiv \sum_{i=1}^{n} b_i x_i r' \equiv \sum_{i=1}^{n} b_i y_i r r' \equiv \sum_{i=1}^{n} b_i y_i \pmod{m}.$$

She can then reconstruct Bob's binary sequence by calling

$$\texttt{SUPERINCREASING_SUBSET_SUM}\ (y_1, \ldots, y_n, s)$$

which returns the set $I = \{i = 1, \ldots, n | b_i = 1\}$.

**Observations** The system is considerably faster than RSA, because it only requires sums, products and modular remainders, no exponentiation.

Since the encryption and decryption processes are not symmetric (Alice cannot use her private key to encrypt something to be decrypted with her public key), the system is not suitable for electronic signature protocols.

Proposed in 1978, in 1984 a polynomial scheme to reconstruct the super-increasing sequence (and hence Alice's private key) was published by Adi Shamir (the "S" in RSA).

Although based on an instantiation of the SUBSET SUM problem, it is commonly referred to as the "Knapsack" cryptosystem.

### 5.5.1 $k$-VERTEX COLORING for $k > 3$

We know that 2-VERTEX COLORING $\in$ **P**, and that 3-VERTEX COLORING is **NP**-complete. What about $k > 3$? Consider, for instance, 4-VERTEX COLORING. On one hand having more colors might seem to relax the problem (more choices mean also more chances of a positive answer); however, we can easily prove that the case $k = 4$ is at least as hard as $k = 3$:

**Theorem 52.** *4-VERTEX COLORING is **NP**-complete.*

*Proof.* Clearly, 4-VERTEX COLORING $\in$ **NP**.

Let us start with a 3-VERTEX COLORING instance $G = (V, E)$ and let us build an equivalent 4-VERTEX COLORING instance $G' = (V', E')$. To build $G'$, let us start from $G$ and add four new

[7] See `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`

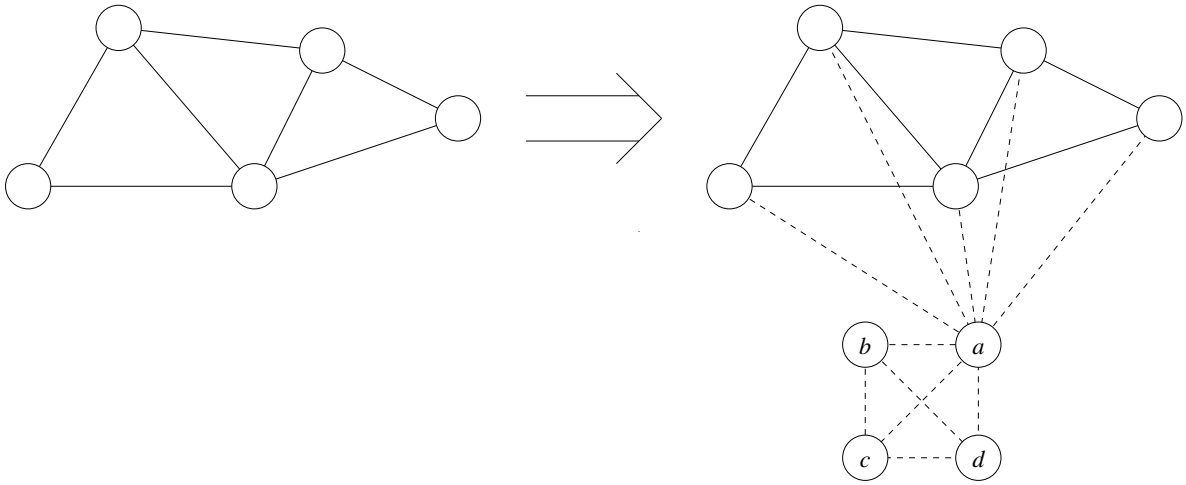



Figure 5.4: Reducing a 3-VERTEX COLORING instance to an equivalent 4-VERTEX COLORING instance.

nodes $a, b, c, d$, all connected to each other (so that every 4-coloring will need to assign different colors to each node). Then, connect $a$ to all nodes of $V$.

If $G$ is 3-colorable, then $G'$ 4-colorable; just assign the fourth color to the extra node $a$.

Conversely, if $G'$ is 4-colorable, then all original nodes in $V$ will have the colors of the extra nodes $b$, $c$ and $d$, therefore they have a valid 3-coloring for the original graph $G$. □

See for example Fig. 5.4: the left-hand side graph is 3-colorable if and only if the right-hand side graph is 4-colorable: the trick consists in wasting the fourth color on node $a$, forcing the remaining nodes to share three colors.

## 5.6 Randomized complexity classes

Observe that the definition of **NP** just requires one non-deterministic computation out of exponentially many to accept the input. Although only one computation might be accepting, there might be better cases in which we are guaranteed that a given fraction of the computations accept the input (if it belongs to the language).

### 5.6.1 The classes RP and coRP

Let us define the following complexity class:

**Definition 50.** *Let $L \in \boldsymbol{NP}$, and let $0 < \varepsilon < 1$. We say that $L$ is* randomized polynomial time*, and write $L \in \boldsymbol{RP}$, if there is a NDTM $\mathcal{M}$ that decides $L$ in polynomial time and, whenever $x \in L$,*

$$\frac{\textit{Number of accepting computations of } \mathcal{M}(x)}{\textit{Number of computations of } \mathcal{M}(x)} \geq \varepsilon. \tag{5.2}$$

Obviously, if $x \notin L$ then there are no accepting computations. In other words, if $L \in \mathbf{RP}$ we are guaranteed that, whenever $x \in L$, a sizable number of computations accept it[8].

**Theorem 53.**

$$\boldsymbol{P} \subseteq \boldsymbol{RP} \subseteq \boldsymbol{NP}.$$

[8]See the Arora-Barak draft, Chapter 7, until Section 7.1 included, for definitions based on "probabilistic Turing Machines"

*Proof.* The second inclusion derives from the definition; for the first one, just observe that a deterministic machine can be seen as a NDTM where all computation coincide, therefore either all computations accept (and the bound (5.2) is satisfied) or all reject. □

Equivalently, if we define **NP** in terms of a deterministic TM $\mathcal{M}$ and polynomial-size certificates $c \in \{0,1\}^{p(|x|)}$, we can define $L \in \mathbf{RP}$ if

$$\frac{|\{c \in \{0,1\}^{p(|x|)} : \mathcal{M}(x,c) = 1\}|}{2^{p(|x|)}} \geq \varepsilon.$$

We can see this definition in terms of probability of acceptance: suppose that $x \in L$, and let us generate a random certificate $c$. Then, $\Pr\big(\mathcal{M}(x,c) = 1\big) \geq \varepsilon$. Conversely, if $x \notin L$ then $\Pr\big(\mathcal{M}(x,c) = 1\big) = 0$, because $x$ has no acceptance certificates.

This fact suggests a method to improve the probability of acceptance at will:

**on input** $x$
**repeat** N **times**
  $c \leftarrow$ random certificate in $\{0,1\}^{p(|x|)}$
  **if** $\mathcal{M}(x,c) = 1$
    **then accept** and **halt**
**reject** and **halt**

In other words, if the machine keeps rejecting $x$ for many certificates, keep trying for $N$ times, where $N$ is an adjustable parameter.

The probability that, given $x \in L$ the machine rejects it $N$ times (and therefore $x$ is finally rejected) is

$$\Pr(\text{REJECT } x | x \in L) \leq (1-\varepsilon)^N.$$

Therefore, by increasing the number $N$ of repetitions, the probability of an error (rejecting $x$ even though $x \in L$) can be made arbitrarily small. Of course, the opposite error (accepting $x$ when $x \notin L$) is not possible because if $x \notin L$ there are no accepting certificates.

This results suggests that the definition of **RP** does not depend on the actual value of $\varepsilon$, as long as it is strictly included between 0 and 1. Observe, in fact, that if $\varepsilon = 0$ then we are not setting any lower bound on the number of accepting computation, and therefore the definition would coincide with that of **NP**, while if $\varepsilon = 1$ then we would require that all computations are accepting, thus rendering the certificate useless, and we would be redefining **P**.

As is customary with classes that are asymmetrical wrt acceptance/rejection mechanisms, we can also define its complementary class **coRP** as the class of languages whose complements are in **RP**, i.e., languages that have an NDTM whose computations always accept $x$ whenever $x \in L$ and such that at least a fraction $\varepsilon$ of computations reject $x$ if $x \notin L$.

### Examples

There are very few "natural" examples of languages in **RP** (or **coRP**) that do not belong to **P** too[9]

**Definition 51.** *Let $f : \{0,1\}^n \to \{0,1\}$ be a computable function mapping $n$-bit strings onto a one-bit value. Then $f$ is* constant *if it has the same value for all inputs:*

$$\forall x, y \quad f(x) = f(y),$$

*while $f$ is* balanced *if it takes both values with equal frequency:*

$$|\{x : f(x) = 0\}| = ||\{x : f(x) = 1\}| = 2^{n-1}.$$

[9]What constituted the main example, PRIMES, is now proved to be in **P**.

Suppose that we are promised that a function $f$ is either constant or balanced. Then, the problem BALANCED FUNCTION is the problem of deciding if $f$ is balanced and it is, in principle, exponential wrt $n$ by virtue of the following algorithm:

**on input** $f$
$f_0 \leftarrow f(0)$
**for** $x \leftarrow \{1, \ldots, 2^{n-1}\}$
  **if** $f(x) \neq f_0$
    **then accept** and **halt**
**reject** and **halt**

In fact, in the worst case we could discover that $f$ is not constant only after evaluating it on half of its possible $2^n$ input values (remember that if $f$ is not constant then it is necessarily balanced).

The following non-deterministic algorithm, on the other hand, is clearly polynomial:

**on input** $f$
Non-deterministically choose $x$,$y \in \{0, \ldots, 2^n - 1\}$
**if** $f(x) = f(y)$
  **then reject**
  **else accept**

However, observe that we are already assured that the function $f$ is either constant or balanced; therefore, if the non-deterministic algorithm accepts (i.e., at least one of its non-deterministic choices leads to acceptance), then it does so with exactly 50% of its computations. If we make some assumptions on the input size (the algorithm decides a function $f$: let's assume that both $f$'s representation on the machine's tape and $f$'s execution time are polynomial wrt $n$), then the algorithm clearly satisfies the rules for **RP**[10].

**Definition 52.** *Let $P = \mathbb{Z}/p\mathbb{Z}[x_1, \ldots, x_n]$ be the ring of polynomials on the finite field $\mathbb{Z}/p\mathbb{Z}$ ($q$ prime). Suppose that $f \in P$ is expressed as a product of low-degree polynomials, e.g.:*

$$f(x_1, \ldots, x_n) = (2x_1 + x_2 - 3x_4 + 2x_6 - 1) \cdot (5x_2 + 4x_3 + x_6 + 2) \cdots (x_2 + 4x_5 + x_{n-1} - 3x_n - 5). \quad (5.3)$$

*The* Polynomial Identity Testing problem *(PIT) is the problem of determining whether $f$ is the zero polynomial or not. In out usual notation,*

$$f \in PIT \quad \leftrightarrow \quad f \equiv 0.$$

Observe that PIT could be decided by writing the polynomial $f$ in canonical form (as a sum of monomials in $x_1, \ldots, x_n$) and verifying that all coefficients are zero. However, transforming the form (5.3) into the canonical form would require an exponential number of multiplications and sums.

The algorithms to decide PIT rely[11] on evaluating $f$ at a number of random points: if any evaluation gives a non-zero value, then $f$ is obviously non-zero; otherwise, there is a (provably bounded) probability of error. In other words, these algorithms always accept $f$ if $f \in$ PIT, but might also accept $f \notin$ PIT with probability bound by a constant $\varepsilon$, which is precisely the definition of **coRP**.

### 5.6.2 Zero error probability: the class ZPP

An interesting characterization of **RP** and **coRP** is the following:

[10] Although this looks like an artificial problem, it is important because it is one of the earliest examples of languages for which quantum machines have an exponential advantage on classical ones, see `https://en.wikipedia.org/wiki/Deutsch-Jozsa_algorithm`

[11] See `https://en.wikipedia.org/wiki/Polynomial_identity_testing` and `https://en.wikipedia.org/wiki/Schwartz-Zippel_lemma` if interested; the PIT problem is also described in Arora-Barak (draft), Section 7.2.2.

- $L \in \mathbf{RP}$ means that there is a machine that, upon random generation of a certificate, never reports false positives (i.e., it only accepts $x$ when $x \in L$), and reports false negatives with probability at most $1 - \varepsilon$;
- $L \in \mathbf{coRP}$ means that there is a machine that, upon random generation of a certificate, never reports false negatives (i.e., it only rejects $x$ when $x \notin L$), and reports false positives with probability at most $1 - \varepsilon$.

If a language $L$ belongs to both **RP** and **coRP**, then it can benefit of both properties. In other words, if $L \in \mathbf{RP} \cap \mathbf{coRP}$, then there are two polynomial-time TMs $M_1$ and $M_2$ and two probability bounds $0 < \varepsilon_1, \varepsilon_2 < 1$ such that

$$\forall x \in \Sigma^* \quad \forall c \in \{0,1\}^{p(|x|)} \qquad \Pr\big(M_1(x,c) \text{ accepts}\big) \text{ is } \begin{cases} 0 & \text{if } x \notin L \\ \geq \varepsilon_1 & \text{if } x \in L \end{cases} \tag{5.4}$$

and

$$\forall x \in \Sigma^* \quad \forall c \in \{0,1\}^{p(|x|)} \qquad \Pr\big(M_2(x,c) \text{ rejects}\big) \text{ is } \begin{cases} 0 & \text{if } x \in L \\ \geq \varepsilon_2 & \text{if } x \notin L. \end{cases} \tag{5.5}$$

We can exploit these two machines with the following algorithm:

**on input** $x$
**repeat**
  $c \leftarrow$ random certificate in $\{0,1\}^{p(|x|)}$
  **if** $\mathcal{M}_1(x,c)$ accepts
    **then accept** and **halt**
  **if** $\mathcal{M}_2(x,c)$ rejects
    **then reject** and **halt**

Observe that this algorithm does not define an explicit number of iterations. However, if $x \in L$, at every iteration $M_1$ has probability $\varepsilon_1$ to accept it, after which the algorithm would stop; conversely, if $x \notin L$, at every iteration $M_2$ has probability $\varepsilon_2$ to reject it, after which the algorithm would stop. with a rejection. If $M_1$ rejects or $M_2$ accepts, we know they they might be wrong and just move on with a new certificate. Therefore, the algorithm will eventually halt, and will always halt with the correct answer.

Suppose that $x \in L$: observe that the number of iterations before halting is distributed as a geometric random variable

$$\Pr(\text{the algorithm makes } n \text{ iterations}) = (1 - \varepsilon_1)^{n-1}\varepsilon_1,$$

whose mean value, representing the expected number of iterations before halting, is

$$E[\text{iterations before halting}] = \frac{1}{\varepsilon_1},$$

which does not depend on anything but the error probability. The same considerations are valid if $x \notin L$.

**Definition 53.** ***ZPP*** $= \boldsymbol{RP} \cap \boldsymbol{coRP}$ *is the class of problems that admit an algorithm that always gives a correct answer and whose expected execution time is polynomial with respect to the input size.*

The following result should be obvious, given the above definition:

**Theorem 54.**

$$\boldsymbol{P} \subseteq \boldsymbol{ZPP} \subseteq \boldsymbol{RP} \subseteq \boldsymbol{NP}.$$

*Proof.* The proof is left as an exercise (exercise 69). □

### 5.6.3 Symmetric probability bounds: classes BPP and PP

Observe that the probabilistic classes shown up to this point are not very realistic: they require an algorithm that never fails for at least one of the two possible answers. Let us define a class that takes into account errors in both senses.

**Definition 54.** *A language $L$ is said to be* bounded-error probabilistic polynomial*, written $L \in \boldsymbol{BPP}$, if there is a NDTM $\mathcal{N}$ running in polynomial time with respect to the input size, such that:*

- *if $x \in L$, then at least 2/3 of all computations accept;*
- *if $x \notin L$, then at most 1/3 of all computations accept (i.e., at least 2/3 of all computations reject).*

In other words, a language is **BPP** if it can be decided by a *qualified* majority of computations of a NDTM. We say that the probability of error is "bounded" precisely because there is a wide margin between the acceptance rate in the two cases.

As usual, the algorithm that emulates the NDTM is built as follows by using the deterministic machine $\mathcal{M}$ that emulates $\mathcal{N}$ via certificates:

**on input** $x$
- $n \leftarrow 0$
- **repeat** N **times**
  - $c \leftarrow$ random certificate in $\{0,1\}^{p(|x|)}$
  - **if** $\mathcal{M}(x, c)$ accepts
    - **then** $n \leftarrow n + 1$
- **if** $n > N/2$
  - **then accept**
  - else **reject**

By making $N$ higher and higher, the probability of error can be reduced at will.

Notice that the 1/3 and 2/3 acceptance thresholds are arbitrary. We just need to have a qualified majority, so an equivalent definition can be given by using any $\varepsilon > 0$ and requiring that the probability of a correct vote (the fraction of correct computations) is greater than $(1/2) + \varepsilon$. In other words, any non-zero separation between the frequencies in the positive and negative case is fine, and provides the same space.

If, on the other hand, we accept *simple majority* votes, then the results are not so nice.

**Definition 55.** *A language $L$ is said to be* Probabilistic polynomial*, written $L \in \boldsymbol{PP}$, if there is a NDTM $\mathcal{N}$ running in polynomial time with respect to the input size, such that:*

- *if $x \in L$, then at least half of all computations accept;*
- *if $x \notin L$, then at most half of all computations accept (i.e., at least half of all computations reject).*

If the frequency of errors can approach 1/2, then the majority might be attained by one computation out of exponentially many, and reaching a predefined confidence level might require an exponential number of repetition ($N$ in the "algorithm" above might not be constant, rather it could be exponential wrt $|x|$).

Given the above definitions, the following theorem should be obvious:

**Theorem 55.**

$$\boldsymbol{RP} \subseteq \boldsymbol{BPP} \subseteq \boldsymbol{PP}.$$

*Proof.* The proof is left as an exercise (exercises 70 and 71). □

The class **BPP** is considered the largest class of "practically solvable" problems, since languages in **BPP** have a polynomial algorithm that, although probabilistic, guarantees an error as small as desired.

No relationship between **NP** and **BPP** is known: it is *unlikely* that $\mathbf{NP} \subseteq \mathbf{BPP}$, because it would imply that all **NP** problems have a satisfactorily probabilistic answer (i.e., heuristics that work very well in all cases); however, the opposite may or may not be the case.

To highlight the impractical nature of **PP**, it is sufficient to show that it contains far too many languages, in particular:

**Theorem 56.** $\boldsymbol{PP \supseteq NP}$

*Proof.* Let us just take the $\mathbf{NP} - complete$ language SATISFIABILITY and provide a machine that accepts satisfiable CNF formulas by simple majority:

**on input** $f$: $n$-variable CNF formula
- $(x_1, \ldots, x_n) \leftarrow$ random truth assignment
- **if** $f(x_1, \ldots, x_n)$
  - **then accept**
  - **else accept** with probability $\frac{1}{2} - \frac{1}{2^{n+1}}$, **reject** otherwise

We can easily check that if $f$ is not satisfiable, then the above algorithm accepts or rejects the input at random, with a very slight bias towards rejection, by a $1/2^{n+1}$ margin against equal odds. This margin is so small that, when $f$ is satisfiable even by just one truth assignment, the $1/2^n$ probability of fortuitously stumbling upon it by chance is enough to tip the probability towards acceptance. In fact, the probability that the algorithm accepts a satisfiable CNF formula $f$ is

$$\begin{aligned}
\Pr(f \text{ accepted} \,|\, f \text{ satisfiable}) &= \Pr(\text{entering "}\mathbf{then}\text{" branch}) \cdot \Pr(\text{acceptance in "}\mathbf{then}\text{" branch}) + \\
&\quad + \Pr(\text{entering "}\mathbf{else}\text{" branch}) \cdot \Pr(\text{acceptance in "}\mathbf{else}\text{" branch}) \\
&\geq \overbrace{\frac{1}{2^n} \cdot 1}^{\text{"}\mathbf{then}\text{" branch}} + \overbrace{\left(1 - \frac{1}{2^n}\right) \cdot \left(\frac{1}{2} - \frac{1}{2^{n+1}}\right)}^{\text{"}\mathbf{else}\text{" branch}} \\
&= \frac{1}{2} + \frac{1}{2^{2n+1}},
\end{aligned}$$

where equality is achieved if there is only one satisfying truth assignment (worst case).

□

The reason why the algorithm described in the above proof is not in **BPP** is the vanishing margin: there is no $\varepsilon > 0$ such that the probability of acceptance is larger that $1/2 + \varepsilon$ for all satisfiable formulas $f$.

Fig. 5.5 and Table 5.1 summarize what has been said in this Section. Observe that there is no known **BPP** algorithm for FACTORING (both positive and negative certificates seem to be exponentially hard to find).

## 5.7 Quantum computing

As already pointed out, the Turing machine is an abstraction encompassing our notion of computability (seen as a sequence of deterministic steps), and our understanding of the relationship between problem complexities has been improved by assuming "enhanced" Turing machines with additional capabilities:

- *non-determinism*, i.e. the capability of "following all computations at once", is definitely not realistic, but allows to easily define significant classes of problems (**NP**, **NEXP**, **NL**...);

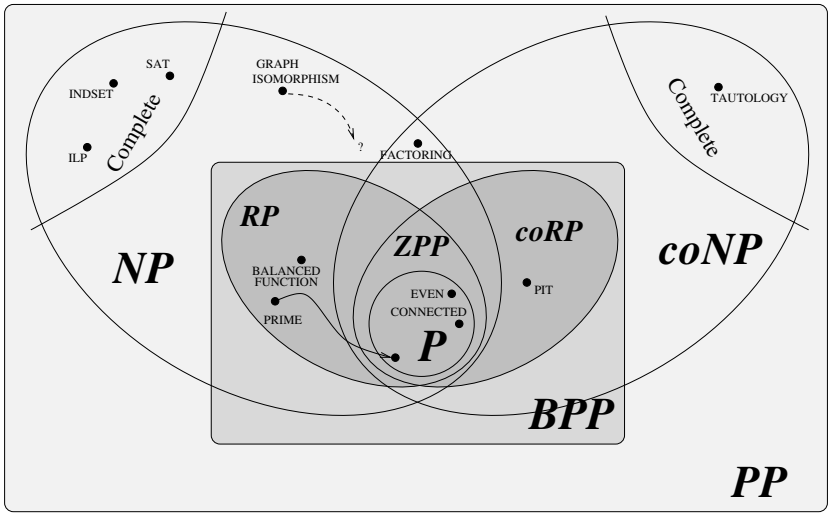



Figure 5.5: What we know about the probabilistic classes introduced in this Section. In particular, the relationship between **BPP** and **NP** is unknown.

Table 5.1: Guaranteed frequency of accepting computations in the various polynomial complexity classes defined in terms of a polynomial NDTM $\mathcal{N}$; $0 < \varepsilon < 1$ is an arbitrary constant value.

| Complexity class | Ratio of accepting computations vs. total computations of $\mathcal{N}(x)$ | | Notes |
|---|---|---|---|
| | if $x \in L$ | if $x \notin L$ | |
| **P** | 1 | 0 | Either all computations accept or all reject; $\mathcal{N}$ might as well be deterministic |
| **NP** | $> 0$ | 0 | One computation out of exponentially many is enough to accept |
| **coNP** | 1 | $< 1$ | Reversed roles of acceptance and rejection |
| **RP** | $> \varepsilon$ | 0 | No false positives; bound probability of false negatives |
| **coRP** | 1 | $< \varepsilon$ | Reversed roles of acceptance and rejection |
| **BPP** | $> 1/2 + \varepsilon$ | $< 1/2 - \varepsilon$ | "Qualified" majority: the $\varepsilon$ margin allows us to reduce error probabilities to arbitrarily small values |
| **PP** | $> 1/2$ | $< 1/2$ | "Simple" majority: no guarantee that error probabilities can be reduced to arbitrary values |

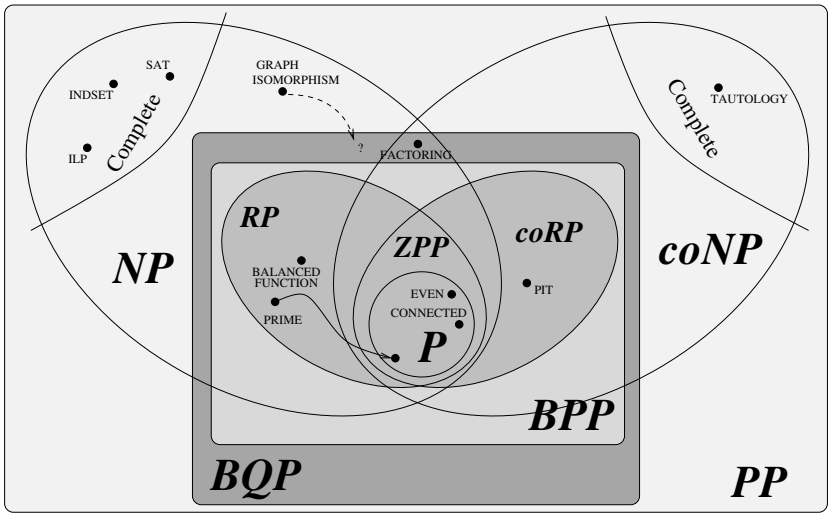



Figure 5.6: Position of the **BQP** quantum class in the previous diagram, with the notable inclusion of the FACTORING language.

- *stochasticity*, i.e. the assumption that the machine can perform random choices, can actually extend the range of problems with an efficient practical solution (up to class **BPP**), but not many problems appear to belong to $\mathbf{BPP} \setminus \mathbf{P}$: while randomness is extensively used in practical settings, it doesn't lead to much improvement when it comes to theoretical worst-case bounds.

Another promising extension to our notion of computation is given by recent advances in *quantum computing*: from time to time, a "quantum" Turing machine can:

- encode a portion of its tape into a physical system ("quantum circuit"), composed by
  - an array of two-state components at quantum scale (quantum bits or "qubits"), and
  - a sequence of transformations ("quantum gates") acting on the qubits;
- evolve the system through the transformations;
- measure the resulting qubit array and encode the measurement outcome onto the tape.

The strength of quantum computing is given by the fact that the status of an $n$-qubit system is actually expressed by an array of $2^n$ complex "amplitudes" (i.e., encoded information is exponential wrt the number of qubits). The weakness of quantum computing is that the amplitude array is not observable, and only determines the probability of a measurement outcome. To this intrinsic weakness we must add lots of engineering problems due to imprecise application of the transformations and to sensibility to external noise.

**Definition 56.** *A quantum algorithm is assumed to be practical when it uses a number of qubits and of quantum gates which is at most polynomial with respect to the input size. If a language is decidable by such quantum algorithm, it is said to belong to class* ***BQP*** *(Bounded-error Quantum Polynomial time).*

Because of the weaknesses reported above, however, very few quantum algorithms are known to actually outperform classical computational models. Integer factorization is the most important problem for which we currently have a polynomial quantum algorithm (Shor's algorithm).

Two significant facts are summarized in Fig. 5.6:

- **BPP** $\subseteq$ **BQP**: quantum systems are true random event generators: any random choice of a stochastic machine can be implemented by a constant-size 1-qubit quantum circuit.
- **BQP** $\subseteq$ **PP**: this is harder to see, however there are ways to emulate quantum circuits with a (possibly exponential) number of classical stochastic computations.

One final note: none of the extensions mentioned above change in any way our notion of computability: the Halting Problem, the Busy Beaver functions, Post's Correspondence Problem, determining the Kolmogorov complexity of a string *remain uncomputable* no matter if we add non-determinism, stochasticity or quantum capabilities. Only their *efficiency* (in terms of time or space) can be improved.

## 5.8 Function problems

In this course we mainly discuss decision problems, aka languages, i.e., questions that require a "yes"/"no" answer. Here is a very brief introduction to function problems.

Let us consider the following "functional" versions of already known problems:

**F-PATH** Given a graph $G = (V, E)$ and two nodes $s, t \in V$, find a path between $s$ and $t$ in $G$.

**F-MINPATH** Given a graph $G = (V, E)$ and two nodes $s, t \in V$, find *a path of minimum length* between $s$ and $t$ in $G$.

**F-SAT** Given an $n$-variable CNF formula $f$, find a satisfying assignment $(x_1, \ldots, x_n)$, provided that it exists.

**F-INDSET** Given a graph $G = (V, E)$ and an integer $k$, find an independent set $V \subseteq V'$ with size $|V'| \geq k$, if any.

Observe that the satisfying assignment might not be unique. In general, we can model a functional problem as a binary relation

$$R \subseteq \Sigma^* \times \Sigma^*.$$

We write $R(x, y)$ —or, equivalently, $(x, y) \in R$— when $x$ is an instance of the problem and $y$ is a corresponding solution.

Two of the problems listed above, F-PATH and F-MINPATH, have obvious polynomial algorithm based on BFS visits of the graph, and are indeed related to languages in **P**. On the other hand, F-SAT and F-INDSET are the functional equivalent of SAT and INDSET, with the "yes"/"no" answer replaced with the request of an actual solution.

The two following definitions extend the notions of **P** and **NP** to functional problems.

**Definition 57 (FP).** *A binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is in class **FP** if there is a polynomial time TM $\mathcal{M}$ that, upon $x \in \Sigma^*$, outputs any $y \in \Sigma^*$ such that $R(x, y)$.*

Note that, given $x$, there may be many $y$ that satisfy the relation (e.g., many truth assignments may satisfy the same CNF formula): we require $\mathcal{M}$ to output one of them.

**Definition 58 (FNP).** *A binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is in class **FNP** if there is a polynomial time TM $\mathcal{M}$ that, upon $x, y \in \Sigma^*$, accepts if and only if $R(x, y)$.*

Observe that we do not actually need non-deterministic machines to define **FNP**: the second member in the relation acts as the certificate[12].

[12] See also the Wikipedia articles about these classes:
`https://en.wikipedia.org/wiki/FP_(complexity)`
`https://en.wikipedia.org/wiki/FNP_(complexity)`

### 5.8.1 Relationship between functional and decision problems

F-SAT and F-INDSET have the following property: if we were able to solve them, then we would automatically have an answer to the corresponding decision problem. I.e., the decision problems have trivial reductions to their functional versions. Therefore, F-SAT and F-INDSET are **NP**-hard.

What about the converse? Suppose that we had an oracle that gives a solution to the decision problem. In both the SAT and INDSET examples, we could use these oracles to build a solution do the functional version step by step. In the F-SAT case, the algorithm would work by guessing the correct truth values one by one:

```
function FSAT (f)                                   f is in CNF
  if SAT(f) = 0                                     if f is unsatisfiable, stop
     then reject and halt
  n ← number of variables of f
  for i ← 1..n
    if SAT(f|_{x_i=⊤}) = 1                          Guess the value of x_i
      then x_i^* ← ⊤                                Put the right truth value in the output string x^*
      else x_i^* ← ⊥
    f ← f|_{x_i=x_i^*}                              Fix x_i in f to the correct truth value and simplify f
  return (x_1^*, ..., x_n^*)
```

Similar methods can be employed also for other problems.

## 5.9 Interactive proof systems

In many cases, a problem's solution is not the only aspect of interest in algorithm research. After a solution is found, the problem remains of "proving" the solution to other actors.

A simple example is captured by the definition of **NP**: even if a super-polynomial solver could find a solution, it could be accepted only if a concise (i.e., polynomial) certificate is provided by the solver. We can model **NP** as the class of languages for which, once a very powerful *prover* (usually called $P$ for prover, or $M$ for Merlin, the wizard in the Arthurian saga) not only needs to provide an answer to a difficult question, but must also provide a proof that can be checked by a less powerful, usually polynomial-time *verifier* (hence called $V$, or $A$ for Arthur[13]).

While, in **NP**, the purpose of Merlin is to convince Arthur when the answer is positive (no certificate is required if the answer is "no"), we can envision more complex cases in which Arthur is more demanding[14].

### 5.9.1 An example: GRAPH ISOMORPHISM

Consider the GRAPH ISOMORPHISM language: given two undirected graphs $G_1 = (V_1, E_1)$ and $g_2 = (V_2, E_2)$, are the two graphs isomorphic, i.e., equal after a permutation of the nodes? The language is clearly in **NP**, but what if we (Arthur) also require a proof of the negative answer —a proof that $G_1$ and $G_2$ are not isomorphic?

Arthur can prepare a sequence of $N$ graphs $G'_1, \ldots, G'_N$, where every $G'_i$ is a random permutation of one of the two original graphs, chosen at random. Arthur sends the graphs $G_1, G_2, G'_1, G'_2, \ldots, G'_N$ to Merlin.

After receiving the graphs, Merlin (who has the capability of solving the problem) has two choices:

1. if $G_1$ is isomorphic to $G_2$, he sends to Arthur the permutation that maps $G_1$ onto $G_2$;

[13] Either Pendragon or Dent — both are fine for our purposes.

[14] For a more comprehensive exposition, see Arora-Barak, first three Sections of Chapter 8 in the online draft.

2. if $G_1$ and $G_2$ are not isomorphic, then it is possible for Merlin to tell which graphs among $G'_1, \ldots, G'_N$ are isomorphic to $G_1$, and which to $G_2$.

In the first case, Arthur can check the permutation and get convinced that the two graphs are indeed isomorphic. In the second case, Arthur checks if Merlin has correctly identified the original graph in all $N$ cases; if $G_1$ and $G_2$ were isomorphic, Merlin could only answer randomly (because all graphs in the set would be isomorphic to both), and the chance for him to be right would be just $2^{-N}$.

Therefore, the described protocol has the following property: if $G_1$, $G_2$ are isomorphic, then Merlin can convince Arthur with certainty; on the other hand, Merlin would have only probability $2^{-N}$ to deceive Arthur by convincing him that they are not. If $G_1$ and $G_2$ are not isomorphic, then Merlin has no way to convince Arthur that they are (he would have to provide a node permutation that works, but there is none); on the other hand, Arthur can be convinced with confidence $1 - 2^{-N}$ of the truth.

### 5.9.2 The Arthur-Merlin protocol

Let us consider, more generally, a protocol where the exchange between Arthur and Merlin requires more than one round. Let $L$ be a language. We assume that Arthur has a polynomial function $f$ that, in order to be convinced whether $x \in L$, generates a "question" $a_1 = f(x)$ for Merlin. We also assume that $f$ is stochastic, in the sense that it can base its output on a finite number of coin tosses.

After receiving the question $a_1$, Merlin uses his own function $g$ (on which we make no assumptions) to generate an "answer" $a_2 = g(x, a_1)$.

The interaction continues for $k$ "question/answer" rounds, where every message may depend on all past history, with the exchange of the following messages:

$$\begin{aligned} a_1 &= f(x) \\ a_2 &= g(x, a_1) \\ a_3 &= f(x, a_1, a_2) \\ a_4 &= g(x, a_1, a_2, a_3) \\ \vdots &\quad \vdots \\ a_{2k-1} &= f(x, a_1, a_2, \ldots, a_{2k-2}) \\ a_{2k} &= g(x, a_1, a_2, \ldots, a_{2k-1}) \end{aligned}$$

After receiving the "final" answer $a_{2k}$, Arthur must decide whether to accept or not the string $x$. Let $\langle f, g \rangle_k(x)$ be the outcome (acceptance or rejection) after the $k$-round interaction.

**Definition 59.** *Given a language $L$ and a positive integer $k$, we say that $L$ is decided by a $k$-round* Merlin-Arthur protocol *($L \in \boldsymbol{AM}[k]$) if there is a stochastic, polynomial function $f$ and a (unbounded) function $g$ such that*

- *if $x \in L$, then $\Pr\big[\langle f, g \rangle_k(x)$ accepts$\big] \geq 2/3$;*
- *if $x \notin L$, then for any function $h$, $\Pr\big[\langle f, h \rangle_k(x)$ accepts$\big] \leq 1/3$.*

In other words, if $x \in L$ then Merlin has a way (function $g$) to convince Arthur to accept with high probability. On the other hand, if $x \notin L$, then whatever method $h$ Merlin uses, he will never be able to convince Arthur to accept with high probability.

### 5.9.3 The Interactive Polynomial protocol class

As an obvious generalization of the $\mathbf{AM}[k]$ class, let us consider a case in which the number of rounds is not constant, but bounded by a polynomial in the size of the input (i.e., we allow for a longer chain of interactions if the input is larger). Definition 59 is modified as follows:

**Definition 60.** *Given a language $L$, we say that $L$ is decided by a* polynomial interactive proof protocol *($L \in$ **IP**) if there is a polynomial $k(n)$, a stochastic, polynomial function $f$ and a (unbounded) function $g$ such that*

- *if $x \in L$, then $\Pr\big[\langle f, g\rangle_{k(|x|)}(x)$ accepts$\big] \geq 2/3$;*
- *if $x \notin L$, then for any function $h$, $\Pr\big[\langle f, h\rangle_{k(|x|)}(x)$ accepts$\big] \leq 1/3$.*

It turns out that **IP** is just another categorization of **PSPACE**. While one inclusion (**IP** $\subseteq$ **PSPACE**) is quite easy to prove, tho other would require too long. Therefore, we just state the theorem without proving it:

**Theorem 57.**

$$\textbf{\textit{IP}} = \textbf{\textit{PSPACE}}.$$

We can also observe that, if we remove all stochasticity from $f$ (i.e., we make Arthur, the prover, deterministic), then Merlin can determine the whole interaction from the beginning, providing the sequence $a_1, \ldots, a_{2k}$ to Arthur since the beginning. Arthur would just need to check his side of the interaction and take a decision without the need of further rounds; therefore, the sequence $a_1, \ldots, a_{2k}$ would be a polynomial certificate of acceptance. Therefore,

**Theorem 58.** *Let **dIP** be the "deterministic" version of **IP** where we strip stochasticity from Arthur's function $f$ in Definition 60. Then*

$$\textbf{\textit{dIP}} = \textbf{\textit{NP}}.$$

Since it is quite universally believed that **NP** $\subsetneq$ **PSPACE**, then we must conclude that, likely, **dIP** $\subsetneq$ **IP**, i.e., stochasticity plays a fundamental role in interactive proofs. This corresponds to our intuition that being able to make random questions increases our chances of discovering a deceit.

## 5.10 Zero-knowledge proofs

In contexts such as **NP** and **IP**, we are used to the idea of *verifiable* answers in the form of a certificate or a proof. Such certificate or proof usually provide a solution to the problem, and the polynomial verifier's task is to check whether the solution is correct or not.

Some contexts, mainly related to security and cryptocurrencies, require one party to "prove" that it knows the answer to a question without disclosing information about it.

One simple example of it are challenge-response algorithms in secure handshake protocols: in order to check if Bob has a cryptographic key, Alice doesn't need to directly ask to see it; she will just challenge Bob to encrypt a random piece of information and she will proceed to compare it to the expected result.

In the following scenario, we will assume that Alice has a graph and needs to 3-color it; Bob has a solution, and he wants to prove it to Alice without disclosing any useful information about the coloring (as he might want to sell his coloring, but Alice needs to be sure that she is getting her money's worth).

Given a graph $G = (V, E)$ with $|V| = n$ nodes, a 3-coloring is a sequence $C = (c_1, c_2, \ldots, c_n) \in \{1, 2, 3\}^n$ (each node is assigned one "color" out of three) where $\{i, j\} \in E \to c_i \neq c_j$.

Observe that each of the $3! = 6$ 3-color permutations gives raise to a different possible coloring. If $\pi : \{1, 2, 3\} \to \{1, 2, 3\}$ is a permutation, then the permuted coloring $\pi C = (\pi c_1, \ldots, \pi c_n)$ is still a valid coloring.

To prove that he has a valid coloring of $G$, Bob will select a cryptographic hash function $H$, a random permutation $\pi : \{1, 2, 3\} \to \{1, 2, 3\}$, $n$ random strings $s_1, \ldots, s_n$ of suitable length (a few hundred bits) and will send to Alice the $n$ values

$$d_i = H(\pi c_i \oplus s_i), \qquad i = 1, \ldots, n, \tag{5.6}$$

where $\oplus$ is the concatenation operator (the bit string representing $\pi c_i$ followed by the bit string $s_i$). Observe that $s_i$ acts as a "salt" string, which randomizes the hashing function.

Once Alice receives $d_1, \ldots, d_n$, she randomly selects one edge $\{i, j\} \in E$ from the graph and sends $i$ and $j$ to Bob, who sends back the two permuted colors $\pi c_i$, $\pi c_j$ and the two salt values $s_i$, $s_j$.

Alice can now check that the two colors are different, that they are in the $\{1, 2, 3\}$ range, and that (5.6) holds for $d_i$ and $d_j$.

We can make the following observations.

- By knowing the $d_i$'s alone, Alice cannot decode the coloring for two reasons:
  1. the hash function $H$ is cryptographically strong, and therefore one would have to try too many combinations of values for $\pi c_i$ and $s_i$ before finding one that returns precisely $d_i$, but more importantly
  2. $H$ is not injective, therefore there might be many combinations that return exactly the same value $d_i$.
- Even after receiving the permuted colors and the salt strings for the two vertices $i$ and $j$, Alice does not get any useful information: all she sees are two different colors, but she has no idea about which permutation was used by Bob, so all combinations are possible (provided that the two colors are different).
- If Bob weren't able to properly 3-color the graph, the colors at the endpoints of at least one edge would be illegal (same color, or out of the allowed range). Since Bob has already sent the hashed values to Alice, and is unable to find a new salt in order to change a node's color depending on Alice's choice, then Alice would discover the problem with probability at least $|E|^{-1}$ (probability of picking the illegal edge, provided that there is at least one).

Therefore, Alice learns nothing about the coloring, and if Bob is cheating then Alice has probability $|E|^{-1}$ to discover it. The probability can be increased at will by repeating the protocol. After $N$ times, the probability for Alice to discover Bob's deceit is at least $1-(1-|E|^{-1})^N$, which can be made arbitrarily close to 1.

Finally, observe that, even though Alice is convinced with very high confidence that Bob knows a 3-coloring for $G$, she cannot show the proof convincingly to another party.

See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more examples of the technique.

# Chapter 6

# Further directions

Here are a few topics that might be interesting and that we could not discuss for lack of time.

## 6.1 About NP

We only analyzed **NP** languages from the (easier) viewpoint of worst-case complexity. A whole line of research is open about *average-case* complexity: given reasonable assumptions on the probability distribution of instances, what is the expected (average) complexity? Even if $\mathbf{P} \neq \mathbf{NP}$, as far as we know the average complexity of some **NP**-complete languages might as well be polynomial. Moreover, for some problems, hard instances might actually exist but be too hard to find. Remember that many applications of **NP**-hardness results (e.g., all public-key cryptography schemes) rely on our ability to actually forge solved instances of hard problems.

Another line of research is *approximability*: for some **NP**-hard optimization (functional) problems, an approximate solution, within a given accuracy bound, might be polynomially achievable.

## 6.2 Above NP

If $\mathbf{PSPACE} \neq \mathbf{NP}$, as is probably the case, there is a very large gap between complete problems in the two classes, and a whole hierarchy of classes, the *polynomial hierarchy*, tries to characterize the enormous (although possibly void!) gap between the two, based on a quantifier-based generalization of the **NP** definition.

## 6.3 Other computational models

Extensions of the Turing model to incorporate quantum mechanics are being studied, and a whole lot of complexity classes (all recognizable because of a Q somewhere in their name[1]) has been proposed. If reliable physical devices will ever be able to implement these quantum models, the solutions to some problems, such as integer factoring, some forms of database search, finding the period of modular functions and so on, will become practical.

[1] But in some cases the "Q" stands for "quantifier" — beware of logicians.

# Part III

# 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 $\Sigma$ 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.1.3 Rice's Theorem

1. Why does the proof of Rice's Theorem fail if the property is *not* semantic (e.g., "the TM has more than 100 states")?

## 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?

### A.2.2 P vs. NP

1. Why is it widely believed that $\mathbf{P} \neq \mathbf{NP}$?
2. Why is it widely hoped that $\mathbf{P} \neq \mathbf{NP}$?

### A.2.3 Other complexity classes

1. Why are classes **EXP** and **NEXP** relatively less studied than their polynomial counterparts?

### A.2.4 General discussion

1. Worst-case complexity might not lead to an accurate depiction of the world we live in. Read Sections 1 and 2 (up to 2.5 inclusive) of the famous "Five worlds" paper:
   Russell Impagliazzo. *A Personal View of Average-Case Complexity.* UCSD, April 17, 1995.
   `http://cseweb.ucsd.edu/users/russell/average.ps`
   What world do we live in, and which would be the ideal world for the Author?

# 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.

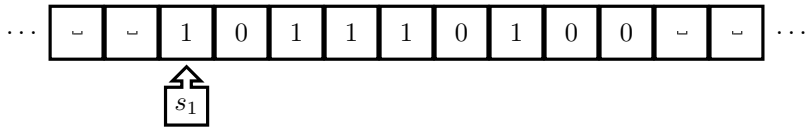
**NOTA BENE** — Solutions to all exercises will be progressively added once they are discussed in class. If you are eager to read solutions, rather than finding them yourself, you you can take a look at test and solution sheets.

**Exercise 1**

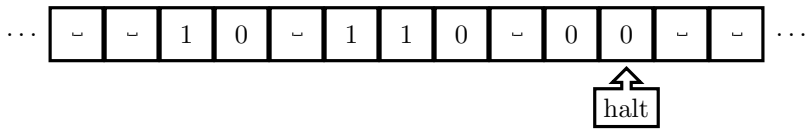
**1.1)** Write a Turing machine according to the following specifications:

- 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 rightmost 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.

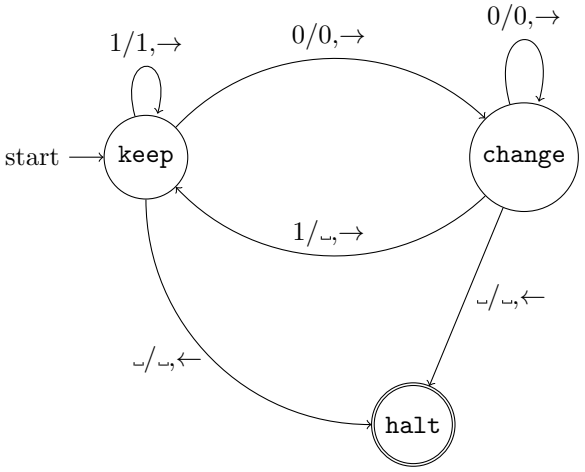
**1.2)** Show the sequence of steps that your machine performs on the input

"010011000111"

**Solution 1**

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` |



**Exercise 2**

Consider the following language on the two-symbol alphabet $\{0, 1\}$:

$$L = \{0^n 1^m \mid n, m \in \mathbb{N} \wedge n > m\}.$$

In plain terms, a string is in $L$ if and only if it starts with a sequence of 0's followed by a (possibly empty) sequence of 1's and nothing else, with strictly more 0's than 1's.
Some examples:

$$\begin{array}{ccc} 00011 \in L & 00111 \notin L & 0 \in L \\ 1 \notin L & 10 \notin L & 11000 \notin L \\ 0110100 \notin L & 0000 \in L & 0011 \notin L \\ & \varepsilon \notin L. & \end{array}$$

Write down a one-tape deterministic Turing Machine $\mathcal{M}$ on the three-symbol alphabet $\{0, 1, \text{␣}\}$ that, given an input string $s \in \{0, 1\}^*$, decides $s \in L$.
You may assume that the input string $s$ is surrounded by infinite blank cells ␣ in both directions, and that the initial current position is the leftmost symbol of $s$.

**Solution 2**

A TM could, for example, keep erasing the leftmost zero and the rightmost one, until only zeroes remain. Any other unexpected condition (zero following a one, no zeroes, and so forth) must cause rejection.
As an example, here is a description of a machine that can be copied and pasted on `http://morphett.info/turing/turing.html`:

```
; Recognize 0^m1^n with m>n.

; Go back and forth, repeatedly removing leftmost 0 and rightmost 1
; until just excess 0's remain. Any other outcome causes rejection.
; The initial state is erase_leftmost_0

; Initially, erase the obligatory leftmost 0
erase_leftmost_0  0 _ r skip_0_right  ; skip all other 0's to the right
erase_leftmost_0  * * r halt-reject   ; Any other symbol, reject

; Keep skipping all 0's to the right
skip_0_right      0 0 r skip_0_right  ; As long as 0, skip it
skip_0_right      1 1 r skip_1_right  ; Upon 1, start skipping 1's
skip_0_right      _ _ l halt-accept   ; All 0's skipped, no 1's: OK

; After all 0's were skipped, start skipping 1's to the right
skip_1_right      1 1 r skip_1_right  ; Keep moving as long as it's 1
skip_1_right      _ _ l erase_rightmost_1 ; move back to erase the last 1
skip_1_right      0 0 r halt-reject   ; After the 1's, there shouldn't be 0's

; Erase the rightmost 1
erase_rightmost_1 1 _ l skip_1_left   ; then start moving left

; Move to the left skipping all 1's
skip_1_left       1 1 l skip_1_left   ; keep skipping 1's
skip_1_left       0 0 l skip_0_left   ; all 1's were skipped, start with 0's
skip_1_left       _ _ l halt-reject   ; no 0's means error

; Keep moving to the left skipping all 0's
skip_0_left       0 0 l skip_0_left   ; as long as there are 0's
skip_0_left       _ _ r erase_leftmost_0 ; passed the whole string, restart
```

Clearly, any description is acceptable, and some missed halting conditions can be forgiven.

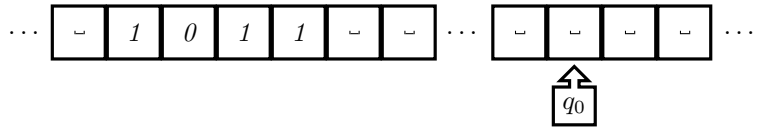
**Exercise 3**

Let $\mathcal{M}$ be a one-tape TM on the alphabet $\Sigma = \{\text{␣}, 0, 1\}$. Assume that the input is a string $s \in \{0,1\}^*$ (i.e., with no blanks in the middle).

**3.1)** Suppose that $\mathcal{M}$ is started on some unknown position (which may fall inside or outside the input string). Define some "preprocessing" states so that the machine finds the leftmost symbol of $s$ before starting its computation.

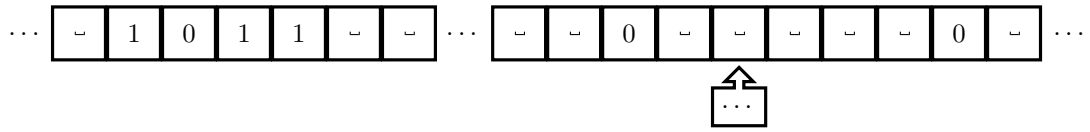
**3.2)** Suppose that the machine starts $d$ cells to the left of the input; how many steps is your solution going to take before it finds the correct initial position? Just provide an asymptotic estimate, e.g. $O(d)$, $O(d^2)\ldots$

Hint — *Observe that we don't know if the initial position of the machine is to the left or to the right of the input string, so we need to perform some back and forth steps. An example initial configuration of our machine might be:*




**Solution 3**

**3.1)** Consider the configuration show in the exercise. The machine will mark a "searched" portion of the tape with two "0" symbols, moving them to extend the searched area:




In the initial state, if the current cell is blank, the machine will put a "0" marker on it, to be interpreted as the left boundary of the searched area, then move to the right to place the corresponding right marker; after placing the right marker the machine will start moving back and forth between the two markers, displacing them by one cell each time:

| | ␣ | 0 | 1 |
|---|---|---|---|
| `begin_input_search` | 0,`set_right_marker`,→ | (see below) | (see below) |
| `set_right_marker` | 0,`search_left_marker`,← | (see below) | (see below) |
| `search_left_marker` | ␣,`search_left_marker`,← | ␣,`set_left_marker`,← | (can't happen) |
| `set_left_marker` | 0,`search_right_marker`,→ | (see below) | (see below) |
| `search_right_marker` | ␣,`search_right_marker`,→ | ␣,`set_right_marker`,→ | (can't happen) |

Cells marked "(see below)" are where the expanding search area "hits" the input string. It may happen in three distinct cases:

1. In the initial state `begin_input_search`, if the current position is on the input; in this case, we have to scan the input to the left until we find the leftmost position, then halt:

| | ␣ | * |
|---|---|---|
| `begin_input_search` | (see above) | *,`find_leftmost_input_symbol`,← |
| `find_leftmost_input_symbol` | ␣,`HALT`,→ | *,`find_leftmost_input_symbol`,← |

   (the 0 and 1 columns are similar, symbols replaced by a * in the table)

2. While expanding the search area to the left, in state `set_left_marker`. In this case, we go erasing the right marker, then we go back (left) to the input and scan it for its leftmost symbol:

| | ␣ | * |
|---|---|---|
| set_left_marker | (see above) | *,search_and_delete_right_marker,→ |
| search_and_delete_right_marker | ␣,search_and_delete_right_marker,→ | ␣,move_left_to_input,← |
| move_left_to_input | ␣,move_left_to_input,← | *,find_leftmost_input_symbol,← |

(note that at the end we can reuse the state `find_leftmost_input_symbol`).

3. Symmetrically, in state `set_right_marker` when we hit the input while moving the right marker:

| | ␣ | * |
|---|---|---|
| set_right_marker | (see above) | *,search_and_delete_left_marker,← |
| search_and_delete_left_marker | ␣,search_and_delete_left_marker,← | ␣,move_right_to_input,→ |
| move_right_to_input | ␣,move_right_to_input,→ | *,find_leftmost_input_symbol,← |

Here is the complete listing for the TM emulator (complete and commented listing with input examples at `https://morphett.info/turing/turing.html?6944cd3645fb0035c6458483d8a468e7`):

```
begin_input_search _ 0 r set_right_marker
set_right_marker _ 0 l search_left_marker
search_left_marker _ _ l search_left_marker
search_left_marker 0 _ l set_left_marker
set_left_marker _ 0 r search_right_marker
search_right_marker _ _ r search_right_marker
search_right_marker 0 _ r set_right_marker

begin_input_search * * l find_leftmost_input_symbol
find_leftmost_input_symbol _ _ r halt
find_leftmost_input_symbol * * l find_leftmost_input_symbol

set_left_marker * * r search_and_delete_right_marker
search_and_delete_right_marker _ _ r search_and_delete_right_marker
search_and_delete_right_marker 0 _ l move_left_to_input
move_left_to_input _ _ l move_left_to_input
move_left_to_input * * l find_leftmost_input_symbol

set_right_marker * * l search_and_delete_left_marker
search_and_delete_left_marker _ _ l search_and_delete_left_marker
search_and_delete_left_marker 0 _ r move_right_to_input
move_right_to_input _ _ r move_right_to_input
move_right_to_input * * l find_leftmost_input_symbol
```

**Exercise 4**

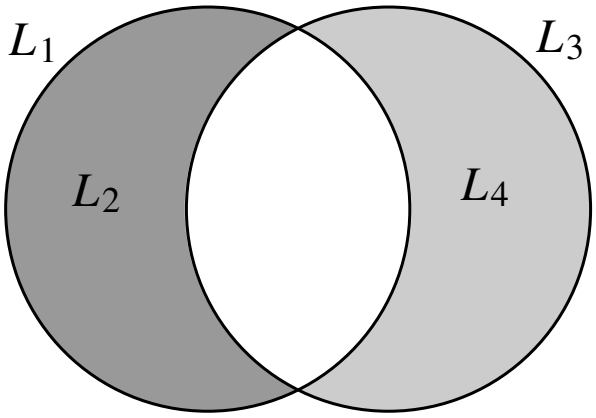
Let TM represent the set of all Turing machines on a given alphabet $\Sigma$, and remember that in our notation $\mathcal{M}(x) = \infty$ means "$\mathcal{M}$ does not halt when executed on input $x$". Consider the following languages:

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

**4.1)** Provide examples of TMs $\mathcal{M}_1, \ldots, \mathcal{M}_4$ such that $\mathcal{M}_1 \in L_1, \ldots, \mathcal{M}_4 \in L_4$.
**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).

**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 (initial state $q_0 = \texttt{HALT}$) is an example for $L_1$ and $L_2$. The machine that never halts (e.g., always moving right and staying in state $q_0$) 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 $L_3$ and $L_4$.
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 where $L_1$ and $L_3$ are represented by the two circles, while $L_2$ and $L_4$ are the two shaded crescents:




A different way to see the relationship is this: $L_1 \cup L_3 = \mathrm{TM}$; TM itself is divided in three: machines that never halt ($L_4$), machines that always halt ($L_2$), and machines that halt for some strings and don't halt for others (the white intersection $L_1 \cap L_3$).

**Exercise 5**

**5.1)** Show that the set $R$ of recursive languages is closed with respect to:

- language union (i.e., if $L_1, L_2 \in R$ then $L_1 \cup L_2 \in R$),
- language intersection (if $L_1, L_2 \in R$ then $L_1 \cap L_2 \in R$) and
- language complement (if $L \in R$ then $\bar{L} = \Sigma^* \setminus L \in R$).

**5.2)** What can be said about the set $RE$ of recursively enumerable languages?

**Solution 5**

**5.1)** If languages $L_1, L_2 \in R$, it means that there are Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$ that decide the two languages:

$$\forall s \in \Sigma^* \qquad \mathcal{M}_1(s) = \begin{cases} 1 & \text{if } s \in L_1 \\ 0 & \text{otherwise} \end{cases} \qquad \mathcal{M}_2(s) = \begin{cases} 1 & \text{if } s \in L_2 \\ 0 & \text{otherwise} \end{cases}$$

In particular, both machines will always halt.

- The following machine will accept $s$ whenever it belongs to $L_1$ *or* $L_2$ and reject it otherwise (hence it decides is $s \in L_1 \cup L_2$):
  - Run $\mathcal{M}_1(s)$; if it accepts, then accept and halt;
  - otherwise, run $\mathcal{M}_2(s)$; if it accepts, then accept and halt;
  - otherwise, since both machines rejected $s$, reject and halt.

  Since the machine decides $L_1 \cup L_2$, language union is still recursive.
- The following machine will accept $s$ whenever it belongs to *both* $L_1$ *and* $L_2$ and reject it otherwise (hence it decides is $s \in L_1 \cap L_2$):
  - Run $\mathcal{M}_1(s)$; if it rejects, then reject and halt;
  - otherwise, run $\mathcal{M}_2(s)$; if it rejects, then reject and halt;
  - otherwise, since both machines accepted $s$, accept and halt.

  The machine decides $L_1 \cap L_2$, intersection is recursive too.
- To decide the complement of language $L_1$, we just need to reverse $\mathcal{M}_1$'s answer:
  - Run $\mathcal{M}_1(s)$; if it accepts, then reject and halt;
  - otherwise, accept and halt.

  This machine decides $\bar{L}_1$ which is therefore recursive.

**5.2)** If languages $L_1, L_2 \in RE$, the Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are not guaranteed to halt if the string is not in the language (they *recognize* it but are not guaranteed to *decide* it):

$$\forall s \in \Sigma^* \qquad \mathcal{M}_1(s) = \begin{cases} 1 & \text{if } s \in L_1 \\ \neq 1 & \text{otherwise} \end{cases} \qquad \mathcal{M}_2(s) = \begin{cases} 1 & \text{if } s \in L_2 \\ \neq 1 & \text{otherwise} \end{cases}$$

where "$\neq 1$" can be interpreted as "reject or run forever."

- To prove closure with respect to language union, we need to alternate steps of both machines, accepting whenever one of the two halts. The following machine will accept $s$ whenever it belongs to $L_1$ *or* $L_2$ (hence it accepts is $L_1 \cup L_2$):

- Initialize the simulation of $\mathcal{M}_1(s)$ and $\mathcal{M}_2(s)$;
- repeat:
  - Run the next step of $\mathcal{M}_1(s)$; if it accepts, then accept and halt;
  - otherwise, run the next step of $\mathcal{M}_2(s)$; if it accepts, then accept and halt;

This machine halts in an accepting state as soon as either $\mathcal{M}_1(s)$ or $\mathcal{M}_2(s)$ accepts, otherwise it runs forever. It accepts $L_1 \cup L_2$, which is therefore recursively enumerable.

- For the intersection, the same construction shown in 5.1 still works. The following machine will accept $s$ whenever it belongs to *both* $L_1$ *and* $L_2$ (hence it accepts $L_1 \cap L_2$):
  - Run $\mathcal{M}_1(s)$; if it rejects, then reject and halt;
  - otherwise, run $\mathcal{M}_2(s)$; if it rejects, then reject and halt;
  - otherwise, since both machines accepted $s$, accept and halt.

  If either $\mathcal{M}_1(s)$ or $\mathcal{M}_2(s)$ doesn't halt, then the machine runs forever, thereby not accepting $s$, as intended. This machine therefore accepts $L_1 \cap L_2$, which is therefore recursively enumerable.
- Finally *RE* is *not* closed under complement. In fact, if both $L$ and $\bar{L}$ are RE, it means that we have both a machine $\mathcal{M}$ that accepts $L$ and a machine $\bar{\mathcal{M}}$ that accepts its complement. Then we could build the following machine that alternates the steps of $\mathcal{M}(s)$ and $\bar{\mathcal{M}}(s)$:
  - Initialize the simulation of $\mathcal{M}(s)$ and $\bar{\mathcal{M}}(s)$;
  - repeat:
    - Run the next step of $\mathcal{M}(s)$; if it accepts, then accept and halt;
    - otherwise, run the next step of $\bar{\mathcal{M}}(s)$; if it accepts, then **reject** and halt;

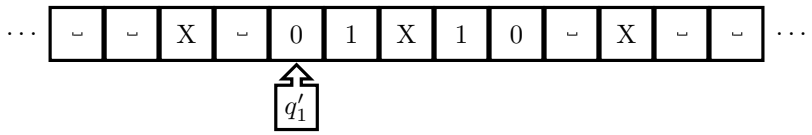
  We are guaranteed that either $\mathcal{M}(s)$ or $\bar{\mathcal{M}}(s)$ will halt and accept; therefore, the constructed machine will either accept $s$ if $s \in L$ or reject $s$ if $s \in \bar{L}$. Therefore it *decides* $L$, which is thus recursive.
  Therefore, closure wrt complement only works for RE languages that are also recursive (and we know that there are RE languages that aren't recursive).

**Observations**

- The only case in which we really need to alternate the execution of the two TMs is the union of RE languages, because either machine could accept. However, alternating the execution of machines also works in the other cases, therefore it isn't an error.
- For the RE case 5.2, it is possible to provide proofs using TMs that enumerate the elements of the language, in place of TMs that recognize them.
- Of course $L_1 \cup L_2 \subseteq L_1$, but even if we know that $L_1$ is recursive, this does not mean that all of its subsets are. For example, $\Sigma^*$ is recursive, but HALT $\subseteq \Sigma^*$ isn't.

**Exercise 6**

Let $\mathcal{M}$ be a one-tape TM on some finite alphabet $\Sigma$, with states $Q = \{q_1, \ldots, q_n\}$ and transition function $f : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \rightarrow\}$.
Suppose now that the tape has some "defective" cells that are marked with a new symbol, X, which is pre-written on them and cannot be modified (i.e., if the machine reads X it can only write X). Our purpose is to create a "robust" version $\mathcal{M}'$ of $\mathcal{M}$.
Of course, the alphabet of $\mathcal{M}'$ will be $\Sigma' = \Sigma \cup \{X\}$. Discuss a systematic way to define a new stateset $Q'$ and a new transition function $f' : \Sigma' \times Q' \to \Sigma' \times Q' \times \{\leftarrow, \rightarrow\}$ that performs the same computation on the defective tape by "skipping" cells marked with the new symbol X.
Assume that the initial input string also skips defective cells; for example, here is a sample initial configuration for input 0110:




**Solution 6**

Every time we find an "X" symbol, we need to leave it as it is, proceed in the same direction as our last move, and remain in the same state, thereby skipping the cell. Therefore, we must encode our last movement direction in the state in order to remember it.
To encode the last direction, our new stateset will be in the form

$$Q' = \{q_1^{\leftarrow}, q_1^{\rightarrow}, \ldots, q_n^{\leftarrow}, q_n^{\rightarrow}\},$$

where every state $q_i$ in $Q$ corresponds to two states $q_i^{\leftarrow}$ and $q_i^{\rightarrow}$. In order to extend the transition function, let $d \in \{\leftarrow, \rightarrow\}$ (so that $q^d \in Q'$ represents one of the two new states generated by the original state $q \in Q$)

$$f'(\sigma, q^d) = \begin{cases} (\sigma', q'^{d'}, d') & \text{if } \sigma \in \Sigma \text{ and } f(\sigma, q) = (\sigma', q', d') \\ (X, q^d, d) & \text{if } \sigma = X. \end{cases}$$

The two states $q_i^{\leftarrow}$ and $q_i^{\rightarrow}$ lead to the same behavior (i.e., the encoded direction is ignored), unless the symbol X is read; in this case, the movement depends on the encoded direction.

**Exercise 7**

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.

**7.1)** TMs with 2 symbols and at most 2 states (plus the halting state), starting from an empty (all-blank) tape.

**7.2)** TMs with at most 100 symbols and 1000000 states.

**7.3)** TMs that only move right;

**7.4)** TMs with a circular, 1000-cell tape.

**7.5)** 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 7**

The following are minimal answers that would guarantee a good evaluation on the test.

**7.1)** 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.

**7.2)** 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.

**7.3)** 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.

**7.4)** 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.

**7.5)** 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 8**

**8.1)** Complete the proof of Theorem 14 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.
**8.2)** Test it for n=3.

**Solution 8**

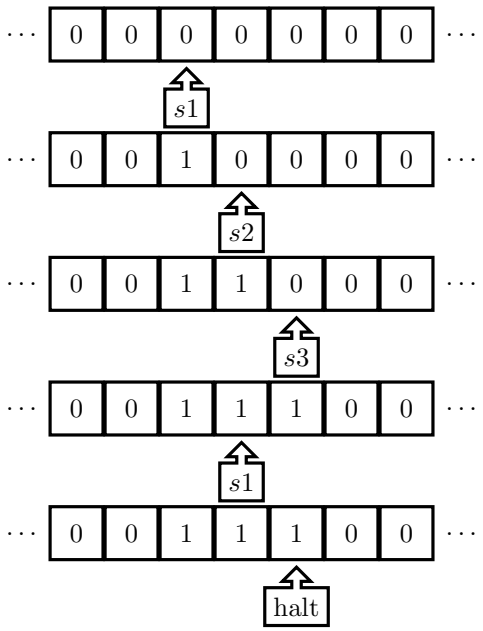
**8.1)** Here is a possible solution:

| | 0 | 1 |
|---|---|---|
| $s_1$ | 1, right, $s_2$ | 1, right, halt |
| $s_2$ | 1, right, $s_3$ | — |
| | ⋮ | |
| $s_i$ | 1, right, $s_{i+1}$ | — |
| | ⋮ | |
| $s_{n-1}$ | 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.
**8.2)** 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 9**

Is it always possible for an instructor to correctly evaluate a student's answer to Exercise 2? Explain.

**Solution 9**

Giving a positive or negative evaluation to the student's answer amounts to deciding the properties "$\mathcal{M}$ decides $L$" and "$\mathcal{M}$ doesn't decide $L$" for the machine $\mathcal{M}$ that he described.
Both properties are semantic, therefore by Rice's Theorem they cannot be decided: there will be some machines for which the instructor won't be able to say whether they correctly answer the question or not.
In other words, the student could embed in the solving machine another TM whose halting property the instructor could not be able to prove.

**Exercise 10**

For each of the following properties of TMs, say whether it is semantic or not, and prove whether it is decidable or not.

**10.1)** $\mathcal{M}$ decides words with an 'a' in them.
**10.2)** $\mathcal{M}$ always halts within 100 steps.
**10.3)** $\mathcal{M}$ either halts within 100 steps or never halts.
**10.4)** $\mathcal{M}$ decides words from the 2018 edition of Webster's English Dictionary.
**10.5)** $\mathcal{M}$ never halts in less than 100 steps.
**10.6)** $\mathcal{M}$ is a Turing machine.
**10.7)** $\mathcal{M}$ decides strings that encode a Turing machine (according to some predefined encoding scheme).
**10.8)** $\mathcal{M}$ is a TM with at most 100 states.

**Solution 10**

**10.1)** 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.*
**10.2)** 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.
**10.3)** 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".
**10.4)** 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 10.1, the language "all words in Webster's"* is *computable, but we aren't able to always decide whether a TM recognizes it or not.*
**10.5)** This is the complement of property 10.2, therefore not semantic and decidable.
**10.6)** 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.
**10.7)** 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.
**10.8)** 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 11**

**11.1)** Prove that the following properties of Turing machines $\mathcal{M}$ are not recursive:

$$\begin{aligned}\mathcal{P}_1 &= \{\mathcal{M} : \mathcal{M}(\varepsilon) \text{ halts after an even number of steps}\}\\ \mathcal{P}_2 &= \{\mathcal{M} : \mathcal{M} \text{ accepts the word “}\texttt{hello}\text{”}\},\end{aligned}$$

where $\varepsilon$ is the empty input string.

**11.2)** Prove that both $\mathcal{P}_1$ and $\mathcal{P}_2$ are recursively enumerable.

Hint — *Point 11.2 can be proved by simulation. How?*

**Solution 11**

**11.1)** Property $\mathcal{P}_1$ is not semantic, therefore Rice's Theorem cannot be invoked.
Since we are dealing with machines running on an empty input, let us reduce the halting problem on empty input (that we called $\text{HALT}_\varepsilon$) to property $\mathcal{P}_1$, and suppose that. Clearly, $\mathcal{P}_1(\mathcal{M}) \Rightarrow \text{HALT}_\varepsilon(\mathcal{M})$. On the other hand, if $\mathcal{P}_1(\mathcal{M})$ is false, $\mathcal{M}$ might still be halting in an odd number of steps. In this case, let us add one dummy step to machine $\mathcal{M}$ before the halting state. Formally:

$$\text{HALT}_\varepsilon(\mathcal{M}) \quad \Leftrightarrow \quad \mathcal{P}_1(\mathcal{M}) \vee \mathcal{P}_1(\mathcal{M} + \text{dummy step before halting}).$$

Otherwise, consider any machine $\mathcal{M}$: if it halts, then the machine $\mathcal{M}+\mathcal{M}$ (consisting of two executions of $\mathcal{M}$) halts in an even number of steps:

$$\text{HALT}_\varepsilon(\mathcal{M}) \quad \Leftrightarrow \quad \mathcal{P}(\mathcal{M} + \mathcal{M}).$$

Property $\mathcal{P}_2$ is clearly semantic and not trivial, hence it is not decidable.

**11.2)** For $\mathcal{P}_1$, consider a UTM that simulates a machine $\mathcal{M}$ and counts the number of steps. If the simulation halts, then $\mathcal{U}$ accepts $\mathcal{M}$ iff the number of steps was even. This machine satisfies the definition of recursive enumerability.
For an explicit enumeration algorithm, let us just employ the usual diagonalization method, but before writing out a halting machine we additionally check whether it halted in an even number of steps.
For $\mathcal{P}_2$, we diagonalize the execution of all TMs on input `hello`; when an execution halts, if the string is accepted then we print the corresponding machine, otherwise we don't. As a result, every machine that accepts `hello` will be eventually printed out.

## Observations

- Again, the direction of the reduction is fundamental. Reducing $\mathcal{P}_1$ to HALT proves nothing, we need to reduce HALT to $\mathcal{P}_1$, i.e., assume that $\mathcal{P}$ is decidable and try to solve HALT with it.
- For the reason above, just saying “in order to decide $\mathcal{P}_1(\mathcal{M})$ we would need to know if $\mathcal{M}$ halts, but this is not possible” doesn't prove anything, because you are only ruling out methods that use HALT to decide $\mathcal{P}_1$; however, that line of reasoning is not excluding other possible ways of deciding $\mathcal{P}_1$ that don't use HALT at all.

**Exercise 12**

Let $\Sigma$ be a finite alphabet, and $L_{\mathrm{R1}}, L_{\mathrm{R2}}, L_{\mathrm{RE1}}, L_{\mathrm{RE2}} \subseteq \Sigma^*$ be four languages on $\Sigma$. $L_{\mathrm{R1}}$ and $L_{\mathrm{R2}}$ are recursive, while $L_{\mathrm{RE1}}$ and $L_{\mathrm{RE2}}$ are recursively enumerable, but not recursive.

**12.1)** For each of the following languages, state if they are recursive, recursively enumerable, or none, and motivate your answers:

- $L_1 = L_{\mathrm{R1}} \cup L_{\mathrm{R2}}$;
- $L_3 = L_{\mathrm{R1}} \cup L_{\mathrm{RE1}}$;
- $L_5 = L_{\mathrm{RE1}} \cup L_{\mathrm{RE2}}$;
- $L_2 = L_{\mathrm{R1}} \cap L_{\mathrm{R2}}$;
- $L_4 = L_{\mathrm{R1}} \cap L_{\mathrm{RE1}}$;
- $L_6 = L_{\mathrm{RE1}} \cap L_{\mathrm{RE2}}$.

**12.2)** State whether the following properties of Turing machines are computable or not, and motivate your statements:

- $\mathcal{P}_1 = \{\mathcal{M} : \mathcal{M} \text{ decides } L_{\mathrm{R1}}\}$;
- $\mathcal{P}_2 = \{\mathcal{M} : \mathcal{M} \text{ decides } L_{\mathrm{RE1}}\}$;
- $\mathcal{P}_3 = \{\mathcal{M} : \text{if } |x| < 100, \text{ then } \mathcal{M} \text{ decides } x \in L_{\mathrm{R1}} \text{ in no more than } |x|^2 + 1 \text{ steps}\}$;
- $\mathcal{P}_4 = \{\mathcal{M} : \text{if } |x| < 100, \text{ then } \mathcal{M} \text{ decides } x \in L_{\mathrm{RE1}} \text{ in no more than } |x|^2 + 1 \text{ steps}\}$.

**Solution 12**

By definition, we can assume that there are TMs $\mathcal{M}_{\mathrm{R1}}$ and $\mathcal{M}_{\mathrm{R2}}$ that respectively decide language $L_{\mathrm{R1}}$ and $L_{\mathrm{R2}}$; likewise, let TMs $\mathcal{M}_{\mathrm{RE1}}$ and $\mathcal{M}_{\mathrm{RE2}}$ recognize languages $L_{\mathrm{RE1}}$ and $L_{\mathrm{RE2}}$ respectively.

**12.1)**

- we can define TM $\mathcal{M}_1$ that decides $x \in L_1$ by checking if $x \in L_{\mathrm{R1}}$ or $x \in L_{\mathrm{R2}}$:

  $$\mathcal{M}_1(x) : \text{if } \mathcal{M}_{\mathrm{R1}}(x) \text{ accepts then accept, else run } \mathcal{M}_{\mathrm{R2}}(x),$$

  therefore $L_1$ is recursive.

- Similarly, we can define TM $\mathcal{M}_2$ that decides $x \in L_2$ by checking if $x \in L_{\mathrm{R1}}$ **and** $x \in L_{\mathrm{R2}}$:

  $$\mathcal{M}_2(x) : \text{if } \mathcal{M}_{\mathrm{R1}}(x) \text{ rejects then reject, else run } \mathcal{M}_{\mathrm{R2}}(x),$$

  thus $L_2$ is recursive too.

- For $L_3$, if $x \notin L_{\mathrm{R1}}$, then we must check if $x \in L_{\mathrm{RE1}}$, but in general $\mathcal{M}_{\mathrm{RE1}}(x)$ only halts if the answer is positive: therefore we can only create a machine that recognizes $L_3$, which is therefore recursively enumerable, but in general not recursive:

  $$\mathcal{M}_3(x) : \text{if } \mathcal{M}_{\mathrm{R1}}(x) \text{ accepts then accept, else run } \mathcal{M}_{\mathrm{RE1}}(x).$$

  Observe, however, that there might be special cases: for instance, if $L_{\mathrm{RE1}} \subset L_{\mathrm{R1}}$, then $L_3 = L_{\mathrm{R1}}$, hence it would be recursive.

- A similar reasoning can be carried out for $L_4$: if $x \in L_{\mathrm{R1}}$, then we must also check if $x \in L_{\mathrm{RE1}}$, but this only halts if the answer is positive: therefore, we can only create a machine that recognizes $L_4$, which is therefore recursively enumerable, but in general not recursive:

  $$\mathcal{M}_4(x) : \text{if } \mathcal{M}_{\mathrm{R1}}(x) \text{ rejects then reject, else run } \mathcal{M}_{\mathrm{RE1}}(x).$$

  Consider, however, the two following special cases (among many possible others):

  - if $L_{\mathrm{R1}} \subset L_{\mathrm{RE1}}$, then $L_4 = L_{\mathrm{R1}}$, hence it would be recursive;
  - if $L_{\mathrm{R1}}$ and $L_{\mathrm{RE1}}$ were disjoint, then $L_4 = \emptyset$, and it would be trivially recursive.

- If both languages are RE, then their union $L_5$ is RE as well, because we can combine the two machines as follows:

  $\mathcal{M}_5(x)$ : alternate the execution of $\mathcal{M}_{\text{RE1}}(x)$ and $\mathcal{M}_{\text{RE2}}(x)$; as soon as one accepts, then accept.

  We need to execute them "in parallel" because either of them might run forever, but we need only one to halt to accept the union language. Again, $L_5$ might as well be recursive.

- Finally, the intersection language $L_6$ can be treated as in the previous intersection cases, since we need both computations $\mathcal{M}_{\text{RE1}}(x)$ and $\mathcal{M}_{\text{RE2}}(x)$ to accept and halt; therefore, $L_6$ is RE.

**12.2)**

- $\mathcal{P}_1$ is semantic and non trivial, therefore undecidable.
- $\mathcal{P}_2$ is trivial: since we explicitly stated that $L_{\text{RE1}}$ is *not* recursive, then no machine can decide it, therefore the property is empty (always false). Hence, $\mathcal{P}_2$ is trivially recursive.
- To assess $\mathcal{P}_3(\mathcal{M})$ for a generic TM $\mathcal{M}$, we "just" need to iterate through all possible strings $x$ of length at most 100 symbols, and simulate the computation $\mathcal{M}(x)$ for at most $|x|^2 + 1$ steps. As soon as a simulations doesn't halt within the allotted number of steps, we reject the property. If the simulation halts within the allotted time, we check its response against the reference machine, by computing $\mathcal{M}_{\text{R1}}(x)$ (which is guaranteed to terminate). If the result is different, we reject. Otherwise, once all strings are tested, we accept. This procedure always terminates with acceptance or rejection, therefore $\mathcal{P}_3$ is recursive.
- The method described for $\mathcal{P}_3$ doesn't work for $\mathcal{P}_4$ because the "reference machine" to which the response of $\mathcal{M}$ must be compared is not guaranteed to halt (since $L_{\text{RE1}}$ is not recursive). However, since we are only interested in strings with less than 100 symbols, we can replace the non recursive language with its finite, length-constrained version:

  $$L'_{\text{RE1}} = \{x \in L_{\text{RE1}} : |x| < 100\}.$$

  Replacing $L'_{\text{RE1}}$ (which is recursive by virtue of its finiteness) for $L_{\text{RE1}}$ in the definition of $\mathcal{P}_4$ does not change the property (since we systematically disregard all longer strings); therefore, $\mathcal{P}_4$ is recursive.

**Observations**

- Even though $L_3$ is a subset of $L_{\text{R1}}$, this fact alone doesn't allow us to conclude that it is recursive, because a subset of a recursive language is not necessarily recursive: think of the language $\Sigma^*$ of all strings, which is trivially computable and contains every other language, e.g., HALT, which is uncomputable.

**Exercise 13**

For each of the following properties:

$$\begin{aligned}\mathcal{P}_1 &= \{\mathcal{M} : \mathcal{M} \text{ accepts the string “0”}\}\\ \mathcal{P}_2 &= \{\mathcal{M} : \mathcal{M} \text{ rejects the string “0”}\}\\ \mathcal{P}_3 &= \{\mathcal{M} : L(\mathcal{M}) \text{ is recursively enumerable}\}\end{aligned}$$

(where $L(\mathcal{M})$ is the language recognized by $\mathcal{M}$)
**13.1)** Prove that the property is / isn't trivial.
**13.2)** Prove that the property is / isn't semantic.
**13.3)** Prove that the property is / isn't recursive.

**Solution 13**

**13.1)** Properties $\mathcal{P}_1$ and $\mathcal{P}_2$ aren't trivial, since there are machines that accept “0” and machines that reject it.
About property $\mathcal{P}_3$, notice that having a TM that recognizes $L$ is the very definition of “recursively enumerable”: every language recognized by a TM is by definition RE. Therefore, $\mathcal{P}_3$ is trivially true for every TM.
**13.2)** Given two TMs $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $L(\mathcal{M}_1) = L(\mathcal{M}_2) = L$, there are two cases: either $0 \in L$, in which case both machines accept it, or $0 \notin L$, thus neither does. Therefore, $\mathcal{P}_1$ is semantic. On the other hand, for property $\mathcal{P}_2$, consider a machine $\mathcal{M}_1$ that rejects all strings (always halts returning 0), and a machine $\mathcal{M}_2$ that never halts; both recognize the same (empty) language: $L(\mathcal{M}_1) = \emptyset = L(\mathcal{M}_2)$; however, $\mathcal{M}_1 \in \mathcal{P}_2$, while $\mathcal{M}_2 \notin \mathcal{P}_2$; therefore, $\mathcal{P}_2$ is not semantic. Finally, $\mathcal{P}_3$ is trivially semantic.
**13.3)** $\mathcal{P}_1$ is non-trivial and semantic; therefore, by Rice's theorem it is non-recursive. $\mathcal{P}_3$ is trivially true for all machines, therefore it is computable by a TM that accepts all strings. Finally, although $\mathcal{P}_2$ is not semantic, we can still follow Rice's Theorem proof to reduce the Halting problem to it (see a very similar case from Exercise 15.4).

**Exercise 14**

Let $L$ be a **finite, non-empty** language on the alphabet $\Sigma = \{0, 1\}$.
For each of the following propositions say whether it is true or false, and briefly motivate your answer.

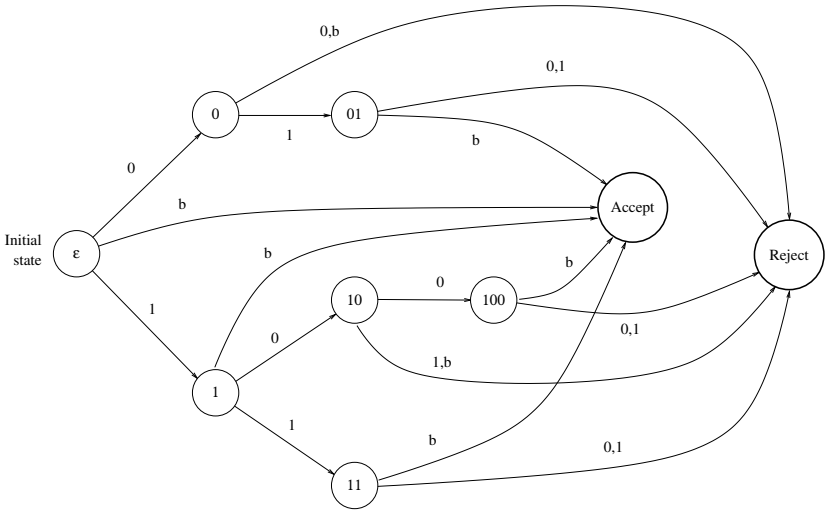
1. $L$ is computable.
2. The property "$\mathcal{M}$ decides $L$," where $\mathcal{M}$ is a deterministic Turing machine, is recursive.
3. The property "the string representing, in binary notation, the number of steps of $\mathcal{M}(\varepsilon)$ before halting belongs to $L$," where $\mathcal{M}$ is a deterministic Turing machine, is recursive.
4. The property "the string representing, in binary notation, the number of states of $\mathcal{M}$ belongs to $L$," where $\mathcal{M}$ is a deterministic Turing machine, is recursive.

**Solution 14**

1. **True** — Since $L$ is finite, it is clearly decidable: a machine just needs to compare the input against a finite number of strings, accepting as soon as a comparison succeeds, or rejecting after all comparisons have failed (a sequence of `if` statements).
   Better yet, the machine may scan the input left to right, encoding the string in its own state, rejecting as soon as it becomes clear that the encoded string is not in $L$, accepting at the end.
   For instance, let $L = \{\varepsilon, 1, 01, 11, 100\}$. Then the following pseudocode clearly decides whether $x \in L$:

   **on input** $x$:
   **if** $x = \varepsilon$ **then accept**;
   **if** $x =$ "1" **then accept**;
   **if** $x =$ "01" **then accept**;
   **if** $x =$ "11" **then accept**;
   **if** $x =$ "100" **then accept**;
   **reject**;

   As an alternative, the following TM decides $L$ by scanning the input and encoding it in the state name as a mnemonic help; as long as the state name is a prefix of some string in $L$ then the machine proceeds; when it finds the first blank, if the state name encodes a string in $L$ then it accepts; in all other cases it rejects:



Notice that the arrows are only marked by the input symbol: the machine always moves right and keeps the same symbols on the tape (it is basically a deterministic finite-state automaton).

2. **False** — "$\mathcal{M}$ decides $L$" is clearly semantic and not trivial, therefore Rice's Theorem applies: the property is undecidable.

3. **True** — The property refers to the behavior of a machine $\mathcal{M}$ on the empty tape, therefore it does not refer to the language recognized by it: it is not a semantic property, thus Rice's Theorem does not apply. Let $m$ be the largest number encoded in binary form by strings in $L$. We just need to simulate machine $\mathcal{M}$ for at most $m$ steps. If it halts in $n$ steps (with $n \leq m$), we just check whether the binary encoding of $n$ is in $L$ (which we can do, because $L$ is computable), otherwise $\mathcal{M}$ clearly doesn't halt within a number of steps encoded in $L$ (maybe it doesn't halt at all). Therefore the property is recursive.
   For instance, if $L = \{\varepsilon, 1, 01, 11, 100\}$ then a machine has the property iff it halts after 1, 3, or 4 steps on the empty tape. Otherwise it doesn't. We just need to simulate $\mathcal{M}$ for at most 4 steps.

4. **True** — Just count the number of states, encode it into $x$ and decide whether $x \in L$.
   If $L = \{\varepsilon, 1, 01, 11, 100\}$, then a machine has the property iff it has 1, 3, or 4 states.

**Observations**

- The sample pseudocode and TM above are just shown for clarification, but there was no need to provide an algorithm for point 1.
  Just the observation that comparing a string against a finite number of alternatives is obviously feasible would achieve full marks.

- Answering "false" to both points 1 and 2 is a significant logical fallacy: if $L$ were uncomputable, then property "$\mathcal{M}$ decides $L$" would be trivial, because no machine has it, and therefore it would be trivially decidable by the machine that always rejects.

- For point 3, we don't need to know if $\mathcal{M}(\varepsilon)$ halts: we just need to simulate it for a maximum number of steps (4 in the example). Therefore, $\text{HALT}_\varepsilon$ is not Turing-reducible to the property.

**Exercise 15**

Let $A \subseteq \Sigma^*$ be a recursive language.
We say that a TM $\mathcal{M}$ "partially decides" $A$ if, for every string $s \in \Sigma^*$, $\mathcal{M}(s)$ either runs forever or provides the correct answer as to $s \in A$. In other words:

$$\forall s \in \Sigma^* \qquad \mathcal{M}(s) = \infty \vee \mathcal{M}(s) = \begin{cases} 0 & \text{if } s \notin A \\ 1 & \text{if } s \in A. \end{cases}$$

**15.1)** Prove that if $\mathcal{M}$ partially decides $A$, then $L(\mathcal{M}) \subseteq A$ (remember that $L(\mathcal{M})$ is the language recognized by $\mathcal{M}$).
**15.2)** Prove that the opposite implication is not true by providing a simple counterexample (suggestion: let $A = \{0, 1\}$; what about a TM that accepts 0 and rejects every other string?)
**15.3)** Prove that the property

$$\mathcal{P} = \{\mathcal{M} : \mathcal{M} \text{ partially decides } A\}$$

is not semantic.
**15.4)** Prove that, even though we cannot invoke Rice's theorem, $\mathcal{P}$ is not recursive (suggestion: try to apply Rice's theorem reduction to $\mathcal{P}$, but pay attention: if a machine $\mathcal{M}$ never halts then $L(\mathcal{M}) = \emptyset$, but $\mathcal{M} \in \mathcal{P}$).

**Solution 15**

**15.1)** By definition, $\mathcal{M}$ will never accept any string that is not in $A$.
**15.2)** Let $A = \{0, 1\}$, as in the suggestion, and consider a TM $\mathcal{M}$ that accepts 0 and rejects all other strings. Clearly $L(\mathcal{M}) = \{0\} \subseteq A$, however $\mathcal{M}$ does not partially decide $A$ because it actively rejects $0 \in A$.
**15.3)** Consider $\mathcal{M}$ from the previous point and a machine $\mathcal{M}'$ that accepts 0 but runs forever for any other input. Clearly, $L(\mathcal{M}) = L(\mathcal{M}') = \{0\}$, but $\mathcal{M}' \in P$, while $\mathcal{M} \notin P$.
**15.4)** Since the machine that never halts clearly is in $P$, in order to use the proof we consider the complementary property

$$\bar{\mathcal{P}} = \{\mathcal{M} : \mathcal{M} \text{ does not partially decide } A\}$$

Then we can precisely follow the same steps of the proof of Theorem 15 at page 24, obtaining a TM $\mathcal{N}'$ that has the property $\bar{\mathcal{P}}$ if and only if the computation $\mathcal{M}_s(t)$ halts, thus reducing the Halting problem to a decision about $\mathcal{N}' \in \bar{\mathcal{P}}$.

**Exercise 16**

For each of the following properties of Turing machines $\mathcal{M}$, prove whether it is recursive or not. Whenever possible, use Rice's theorem.

**16.1)** $\mathcal{M}$ either performs less than 100 steps or runs forever when executed on an empty tape;

**16.2)** $\mathcal{M}$ never visits any state more than ten times when executed on an empty tape;

**16.3)** $\mathcal{M}$ recognizes Turing machines with more states than alphabet symbols.

**Solution 16**

**16.1)** Non-recursive. The property is not trivial because clearly there are machines with that property and machines without it, however it is not semantic (e.g., a machine might recognize the empty language and reject immediately, or run 101 dummy states and then reject), therefore we cannot use Rice's theorem. We could use a TM computing $\mathcal{P}_1$ to test for $\mathcal{M} \in \mathrm{HALT}_\varepsilon$ in two ways:

- create $\mathcal{M}'$ by adding 100 dummy states at the beginning of $\mathcal{M}$, so that $\mathcal{M}'$ must run for at least 100 steps and behaves exactly like $\mathcal{M}$ in every other aspect, then test $\mathcal{M}' \in \mathcal{P}_1$;
- or test $\mathcal{M} \in \mathcal{P}_1$ and, if yes, simulate a run of $\mathcal{M}(\varepsilon)$ for at most 100 steps to see whether it halts within 100 steps; if not, it will run forever.

**16.2)** Recursive. Again, the property is neither trivial nor semantic, so Rice's Theorem cannot be applied. However, to check whether $\mathcal{M} \in \mathcal{P}_2$ we just need to maintain a counter for every state of $\mathcal{M}$ and simulate the computation $\mathcal{M}(\varepsilon)$ increasing a counter whenever the computation visits the corresponding state. As soon as one counter exceeds 10, we reject (if $\mathcal{M}$ runs forever, we are guaranteed that this will eventually happen, because the number of states is finite). If the computation halts before any counter exceeds 10, then we accept.

**16.3)** Non-recursive. The definition clearly defines a language (the actual meaning of the definition is "$\mathcal{M}$ recognizes the language of all TM descriptions that…"), but it is not trivial (it is possible to build a TM with the property of recognizing TMs with more states than symbols). Rice's theorem applies.

**Exercise 17**

For each of the following properties of a Turing machine $\mathcal{M}$, prove whether it is computable or not. When possible, invoke Rice's Theorem.
**17.1)** $\mathcal{M}$ accepts at least one input string.
**17.2)** $\mathcal{M}$ accepts at least one input string of length 3.
**17.3)** $\mathcal{M}$ accepts at least one input string of length 3 within 1000 steps.
**17.4)** $\mathcal{M}$ accepts at least one input string within 1000 steps.

**Solution 17**

**17.1)** The property is clearly semantic: if two TMs decide the same language $L$, they both have the property (if $L \neq \emptyset$) or neither has it (if $L = \emptyset$). The same observation proves that the property is not trivial. Therefore, Rice's Theorem applies, and the property is not computable.
**17.2)** Same as above, where the discriminant is whether $L$ contains at least one 3-symbol string or not.
**17.3)** The property, while not trivial, isn't semantic: take a TM $\mathcal{M}$ that accepts a 3-symbol string in few steps (thereby having the requested property) and modify it into $\mathcal{M}'$ by adding 1000 dummy steps before accepting, so that $L(\mathcal{M}) = L(\mathcal{M}')$; however, $\mathcal{M}'$ takes too many steps and doesn't have the property. Therefore Rice's Theorem doesn't apply.
The property is computable: to check whether $\mathcal{M}$ has it, we just simulate it for at most 1000 steps on every 3-symbol string (a finite set). If $\mathcal{M}$ ever accepts an input within the step limit, then we conclude that it has the property; if we exhaust all strings without reaching acceptance, we conclude that $\mathcal{M}$ doesn't.
**17.4)** Same as above. Although we apparently have no limit on the accepted string size, we know that at most the first (or last, depending on the initial position of the machine) 1000 symbols will ever affect a decision within the allowed number of steps, so we really need to simulate the machine on all 1000-symbol inputs.

**Remarks**

- For the last two points, the two key observations are:
  - we need to check all possible input string, because we don't know which one is going to work;
  - however, the number of strings to be tested is finite, therefore our search always ends.

  Without these explicit observations, the answer is considered incomplete and cannot receive full marks.
- the fact that a property doesn't meet the conditions of Rice's Theorem doesn't mean that it is computable.

**Exercise 18**

Consider the following language in $\{0,1\}^*$:

$$K = \{0^n 1^n : n \in \mathbb{N}\} = \{\varepsilon, 01, 0011, 000111, 00001111, 0000011111, \dots\}$$

i.e., all strings composed by a sequence of zeroes followed by the *same* number of ones.
**18.1)** Write a single-tape Turing Machine with alphabet $\Sigma = \{\text{␣}, 0, 1\}$ that recognizes $K$.
**18.2)** Prove or disprove the decidability of each of the following properties of TMs:

$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} : \mathcal{M} \text{ decides } K\},\\
\mathcal{P}_2 &= \{\mathcal{M} : \mathcal{M} \text{ decides } K \text{ in less than 100 steps}\},\\
\mathcal{P}_3 &= \{\mathcal{M} : \mathcal{M} \text{ decides } K \cap \Sigma^{100} \text{ (i.e., strings in } K \text{ not longer that 100 symbols)}\}.
\end{aligned}$$

Hint — *For 18.1 use any notation you like, and encode acceptance and rejection as you prefer (0/1 on tape, two different halting states, etc.).*

**Solution 18**

**18.1)** The simplest, although, not the most efficient, machine just keeps erasing the leftmost 0 and the rightmost 1 until the input is empty or some unexpected symbol appears (e.g., leftmost 1, rightmost 0, blank when a 1 should be erased).
We assume that the input is a contiguous string of 0's and 1's, surrounded by blanks, and that the machine starts on the leftmost input symbol. Here is the transition table:

| | ␣ | 0 | 1 |
|---|---|---|---|
| `erase-leftmost-0` | ␣/→/`accept` | ␣/→/`go-right` | 1/→/`reject` |
| `go-right` | ␣/←/`erase-rightmost-1` | 0/→/`go-right` | 1/→/`go-right` |
| `erase-rightmost-1` | ␣/←/`reject` | 0/←/`reject` | ␣/←/`go-left` |
| `go-left` | ␣/→/`erase-leftmost-0` | 0/←/`go-left` | 1/←/`go-left` |

An encoding suitable for the TM simulator seen in class[1] is:

```
erase-leftmost-0 0 _ r go-right    ; found and erased a 0
erase-leftmost-0 1 1 r halt-reject ; unexpected 1
erase-leftmost-0 _ _ r halt-accept ; the input has been consumed

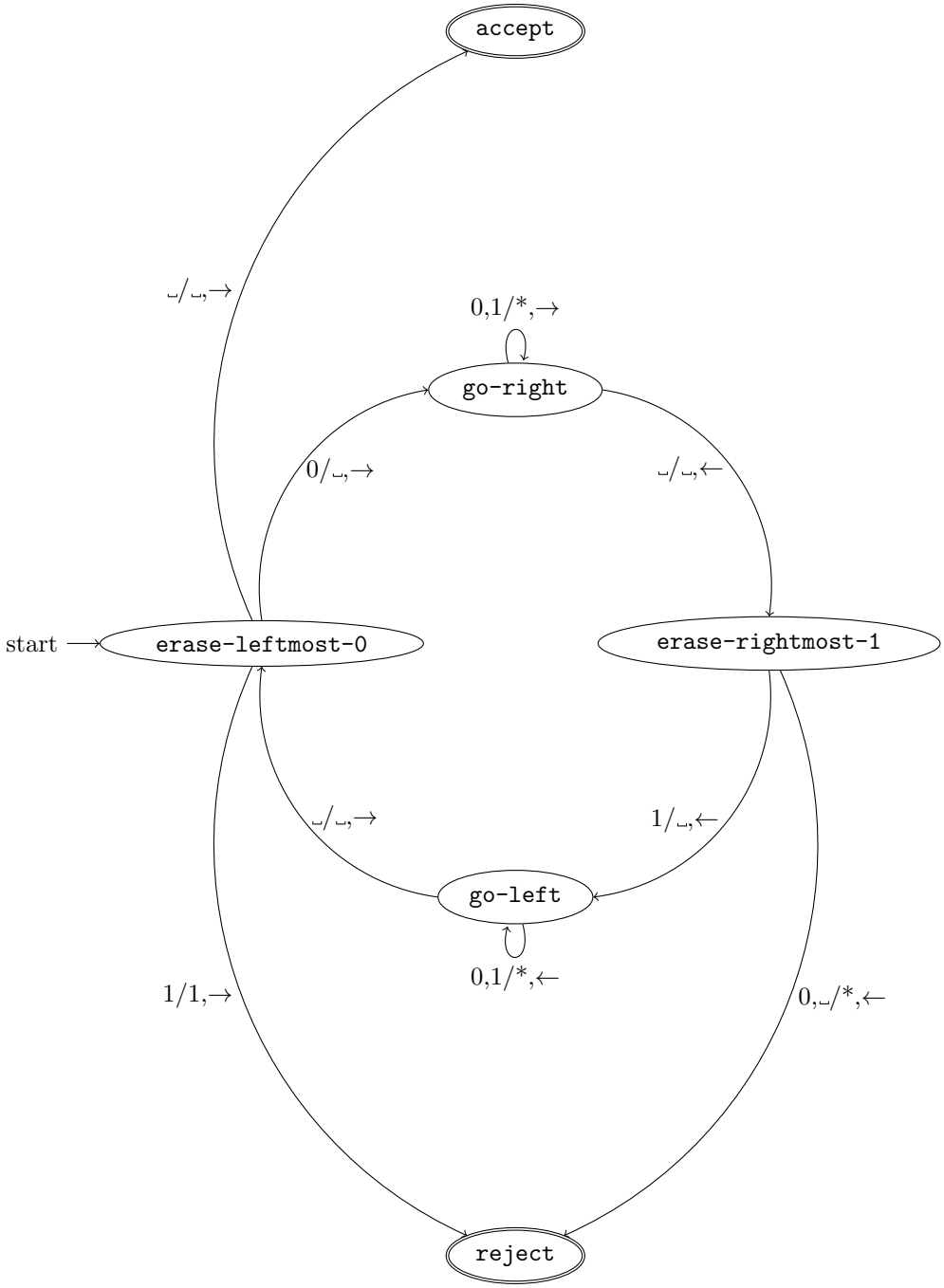
go-right 0 0 r go-right          ; keep skipping the input
go-right 1 1 r go-right
go-right _ _ l erase-rightmost-1 ; found the end of the input

erase-rightmost-1 1 _ l go-left     ; found and erased a 1
erase-rightmost-1 0 0 l halt-reject ; unexpected 0
erase-rightmost-1 _ _ l halt-reject ; unexpected blank

go-left 0 0 l go-left          ; keep skipping the input
go-left 1 1 l go-left
go-left _ _ r erase-leftmost-0 ; found the beginning of the input
```

The same machine as an automaton form:

[1] `http://morphett.info/turing/turing.html`




**18.2)**

- Property $\mathcal{P}_1$ is clearly semantic ($\mathcal{M} \in \mathcal{P}_1 \Leftrightarrow L(\mathcal{M}) = K$) and is not trivial (there is at least one machine that decides $K$ and at least one that doesn't); therefore, by Rice's Theorem, it is undecidable.
- A TM limited to 100 steps cannot decide $K$. Consider, e.g., the string $s_1 = 0^{1000}1^{1000} \in K$. A TM limited to 100 steps wouldn't be able to read the whole input, therefore it wouldn't be able to tell $s_1$ from $s_2 = 0^{1000}1^{1001} \notin K$. Therefore, $\mathcal{P}_2 = \emptyset$, hence it is trivially computable by a TM that always rejects.

- Again, $\mathcal{P}_3$ is semantic and non-trivial, thus uncomputable.

## Observations

- Many other TMs are possible for 18.1.
- Observe that, since 18.1 requires the TM to just *recognize* $K$, rejection could be replaced by a non-halting computation.
- As usual, there is a significant distinction between the computability of $K$ and the computability of the property "This machine decides $K$".
- Property $\mathcal{P}_2$ doesn't just require the TM to halt after 100 steps, but also to decide $K$. Therefore, simulating the TM for 100 steps isn't enough: we also need to consider which inputs it should be simulated on.
- The fact that the language defining $\mathcal{P}_3$ is finite doesn't matter: Rice's theorem is still valid, because we wouldn't be able to always assert whether a TM would halt or not.

**Exercise 19**

In the following property definitions, a finite alphabet $\Sigma$ is given; $\mathcal{M}$ spans all DTMs on $\Sigma$; $x \in \Sigma^*$ spans all strings; finally, $\mathcal{M}(x)$ represents the computation of $\mathcal{M}$ on input $x$:

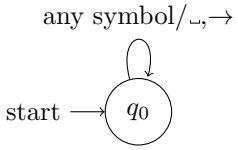
$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} : \forall x\ \mathcal{M}(x) \text{ leaves the initial state } q_0 \text{ at the first step}\} \\
\mathcal{P}_2 &= \{\mathcal{M} : \forall x\ \mathcal{M}(x) \text{ never enters the initial state } q_0 \text{ after the first step}\} \\
\mathcal{P}_3 &= \{\mathcal{M} : \forall x\ \mathcal{M}(x) \text{ changes state at every step}\}.
\end{aligned}$$

**19.1)** Prove that none of the above properties is semantic.
**19.2)** For each of the properties above, prove whether it is computable or not.
**19.3)** Prove or disprove the following statement:

*Every trivial property of Turing machines is semantic.*

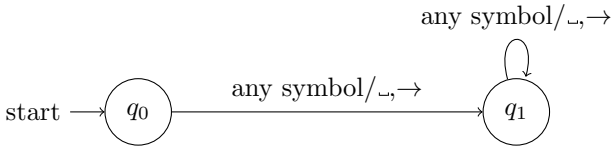
**Solution 19**

**19.1)** By definition, a machine that runs forever accepts the empty language. Let $\mathcal{M}_1$ be a machine that keeps going left remaining in the same state:



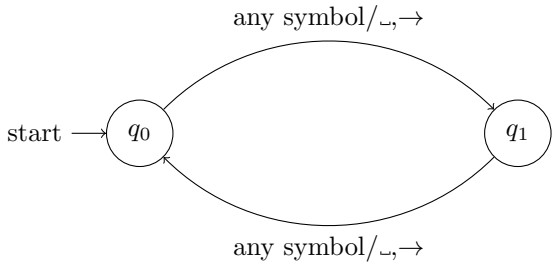

| | 0 | 1 | ␣ |
|---|---|---|---|
| $q_0$ | $\text{␣}, q_0, \rightarrow$ | $\text{␣}, q_0, \rightarrow$ | $\text{␣}, q_0, \rightarrow$ |

Let $\mathcal{M}2$ be a machine that in the first step always transition to a new state, then keeps running forever:




| | 0 | 1 | ␣ |
|---|---|---|---|
| $q_0$ | $\text{␣}, q_1, \rightarrow$ | $\text{␣}, q_1, \rightarrow$ | $\text{␣}, q_1, \rightarrow$ |
| $q_1$ | $\text{␣}, q_1, \rightarrow$ | $\text{␣}, q_1, \rightarrow$ | $\text{␣}, q_1, \rightarrow$ |

Clearly, $L(\mathcal{M}_1) = \emptyset = L(\mathcal{M}_2)$, however $\mathcal{M}_1 \notin \mathcal{P}_1$, while $\mathcal{M}_2 \in \mathcal{P}_1$. Therefore, $\mathcal{P}_1$ is not semantic. The same machines also work for $\mathcal{P}_2$, since $\mathcal{M}_1$ keeps reentering the initial state, while $\mathcal{M}_2$ doesn't. Observe that neither $\mathcal{M}_1$ nor $\mathcal{M}_2$ change state at every step, therefore neither has property $\mathcal{P}_3$. However, we can build a third machine, $\mathcal{M}_3$, that keeps changing state while running forever:




| | 0 | 1 | ␣ |
|---|---|---|---|
| $q_0$ | $\text{␣}, q_1, \rightarrow$ | $\text{␣}, q_1, \rightarrow$ | $\text{␣}, q_1, \rightarrow$ |
| $q_1$ | $\text{␣}, q_0, \rightarrow$ | $\text{␣}, q_0, \rightarrow$ | $\text{␣}, q_0, \rightarrow$ |

Again, $L(\mathcal{M}_3) = \emptyset$, however (unlike $\mathcal{M}_1$ and $\mathcal{M}_2$) $\mathcal{M}_3 \in \mathcal{P}_3$. Therefore, $\mathcal{P}_3$ is not semantic either.
**19.2)** $\mathcal{P}_1$ is computable: to decide it, we just need to look at the transition function and check that, no matter the input symbol, the rules for $q_0$ always lead to a different state.

The same idea doesn't work for $\mathcal{P}_2$, since we also need the machine to never go back to $q_0$; even if there is some rule that leads back to $q_0$, we need to check if it's ever used. Indeed, the property is non-recursive, and we can reduce the Halting problem (in the version with the empty input, $\text{HALT}_\varepsilon$) to it.

Suppose that we are given a TM $\mathcal{M}$, and we want to know if $\mathcal{M}$ ever halts. We can tweak it into a new machine $\mathcal{M}'$ by adding a dummy initial state $q_0'$ that does nothing but immediately move to a new state $q_1'$ that erases the tape (in this way, we ensure that $\mathcal{M}'$ satisfies $\mathcal{P}_2$). Once the tape is erased, $\mathcal{M}'$ transitions to the initial state of $\mathcal{M}$ and starts computing $\mathcal{M}(\varepsilon)$ (because the tape is now empty). However, we replace all transitions to the halting state with transitions to $q_0'$. Therefore, $\mathcal{M}'(x)$ immediately leaves its initial state, then follows the same steps as $\mathcal{M}(\varepsilon)$; however, if $\mathcal{M}(\varepsilon)$ halts then $\mathcal{M}'(x)$ returns to state $q_0'$, thereby violating property $\mathcal{P}_2$. To summarize, $\mathcal{M}(\varepsilon)$ halts if and only if $\mathcal{M}'(x)$ returns to its original state, which is a different way of saying that $\mathcal{M}(\varepsilon)$ halts if and only if $\mathcal{M}' \notin \mathcal{P}_2$. We have therefore reduced $\text{HALT}_\varepsilon$ to $\mathcal{P}_2$, thereby proving that it is uncomputable.

A similar construction lets us prove that $\mathcal{P}_3$ is uncomputable. Again, we reduce $\text{HALT}_\varepsilon$ to $\mathcal{P}_3$. Let $\mathcal{M}$ be a TM; Let us transform it to a new machine $\mathcal{M}'$ that initially erases it input and then executes the original $\mathcal{M}$ on the empty tape. To make property $\mathcal{P}_3$ valid, we double the number of states of $\mathcal{M}'$ by inserting an "even" and an "odd" version of each state, and making "even" states transition to "odd" ones and vice versa, so that $\mathcal{M}'$ actually changes state at every step, satisfying property $\mathcal{P}_3$. Finally, whenever $\mathcal{M}$ halts, let $\mathcal{M}'$ remain in the same state, thereby violating $\mathcal{P}_3$. Therefore, $\mathcal{M}(x)' \in \mathcal{P}_3$ if and only if $\mathcal{M}(\varepsilon)$ never halts, and we have reduced $\text{HALT}_\varepsilon$ to $\mathcal{P}_3$.

## Observations

- The fact that the statement of a property mentions states or other syntactic elements doesn't automatically mean that the property is not semantic. Just stick to the definition.
- $\mathcal{P}_3$ does not require the TM to visit a *new* state at each step. If so, the property would be computable, because we just need to check if the machine halts before exhausting all states (and this can be checked for all inputs, because only a finite portion of them can be scanned).
- "To verify the property we need to simulate the machine" is *always* a deeply wrong answer. Simulation is just one out of many possible ways to analyze an algorithm.

**Exercise 20**

For each of the following properties of Turing machines, show whether they are recursive or not. Invoke Rice's Theorem when possible; if it doesn't apply, explain why.

$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} | \mathcal{M} \text{ halts in at most 100 steps on the empty tape}\};\\
\mathcal{P}_2 &= \{\mathcal{M} | \mathcal{M} \text{ halts in at most 100 steps on every input}\};\\
\mathcal{P}_3 &= \{\mathcal{M} | \mathcal{M} \text{ halts on every input}\};\\
\mathcal{P}_4 &= \{\mathcal{M} | L(\mathcal{M}) \text{ only contains strings of even length}\}.
\end{aligned}$$

**Solution 20**

- $\mathcal{P}_1$ isn't semantic: two TMs may recognize the same language, yet have totally different runtimes on the empty string, just by adding dummy states at the beginning or at the end of the computation. Therefore Rice's Theorem doesn't apply.
  The property is computable: to test $\mathcal{P}_1(\mathcal{M})$ we just need to simulate the first 100 steps of $\mathcal{M}(\varepsilon)$. If it halts within the simulation then the property is true, else it isn't.

- $\mathcal{P}_2$ isn't semantic for the same reason as $\mathcal{P}_1$.
  Again, the property is computable. Although the property requires testing "every input" of the machine, since the simulation only looks for 100 steps we only need to test $\mathcal{M}(s)$ it for all possible combinations $s$ of symbols in the 201 tape cells that are reachable within the time limit (100 to the right and 100 to the left): the remaining cells, albeit infinite, have no influence on the simulation.

- $\mathcal{P}_3$ is semantic: as soon as we define "acceptance" as halting (i.e., if $\mathcal{M}(s)$ halts then $\mathcal{M}$ accepts $s$), then the property means that $\mathcal{M}$ accepts every input, i.e., $L(\mathcal{M}) = \Sigma^*$ (where $\Sigma$ is $\mathcal{M}$'s alphabet). The property is not trivial (there are plenty of machines that accept all strings, and plenty that don't). Hence, Rice's Theorem is applicable and $\mathcal{P}_3$ is not computable.
  As analternative, if we don't want to invoke the "semanticity" of $\mathcal{M}$, we can reduce $\text{HALT}_\varepsilon(\mathcal{M})$ to $\mathcal{P}_3$: given $\mathcal{M}$, transform it into $\mathcal{M}'$ that erases it input and then executes $\mathcal{M}$. Clearly, $\mathcal{M}'$ halts on every string if and only if $\mathcal{M}$ halts on the empty string:

  $$\text{HALT}_\varepsilon(\mathcal{M}) \Leftrightarrow \mathcal{P}_3(\mathcal{M}').$$

  We have therefore shown that $\text{HALT}_\varepsilon \leq_\text{T} \mathcal{P}_3$, meaning that $\mathcal{P}_3$ is "less" computable that the halting problem.

- $\mathcal{P}_4$ is clearly semantic, because it refers to the language accepted by $\mathcal{M}$. More specifically, if $L(\mathcal{M}_1) = L(\mathcal{M}_2) = L$ then either $L$ only has even-sized strings (thus $\mathcal{P}_4(\mathcal{M}_1) = \mathcal{P}_4(\mathcal{M}_2) = \text{true}$) or it doesn't (thus $\mathcal{P}_4(\mathcal{M}_1) = \mathcal{P}_4(\mathcal{M}_2) = \text{false}$). In both cases, $\mathcal{P}_4(\mathcal{M}_1) = \mathcal{P}_4(\mathcal{M}_2)$. $\mathcal{P}_4$ is clearly not trivial, hence Rice's Theorem applies and $\mathcal{P}_4$ is not computable.

**Observations**

For $\mathcal{P}_3$ it is not enough to say "to see if $\mathcal{M}$ halts on every string we should test the halting problem on infinite strings, which is not feasible". This reasoning only considers *one* possible way of solving the problem (namely, using $\text{HALT}(\mathcal{M}, s)$), but others could exist. In other words, the reduction is performed in the wrong way, $\mathcal{P}_3 \leq_\text{T} \text{HALT}$, while the correct sense to show $\mathcal{P}_3$'s uncomputability would be $\text{HALT} \leq_\text{T} \mathcal{P}_3$, as in the proposed solution.

**Exercise 21**

Let $\Sigma = \{\texttt{0}, \texttt{1}\}$ be the set of binary digits and consider the language $L \subset \Sigma^*$ that contains all and only binary strings with alternating `0` and `1` (i.e., every `0` is followed by `1` and vice versa) with at most one exception (repeated digits `00` or `11`). For instance, the following strings belong to $L$ because there is at most one repetition (underlined):

$$\texttt{0} \quad \underline{\texttt{11}} \quad \texttt{01} \quad \texttt{0}\underline{\texttt{0}}\texttt{1} \quad \texttt{1010101} \quad \texttt{1010}\underline{\texttt{11}}\texttt{01} \quad \underline{\texttt{00}}\texttt{10101} \quad \texttt{1}\underline{\texttt{00}}\texttt{101010} \quad \varepsilon.$$

The following don't (they contain more than one pair of adjacent equal digits, `00` or `11`):

$$\underline{\texttt{0011}} \quad \texttt{01}\underline{\texttt{00}}\texttt{10101}\underline{\texttt{00}}\texttt{1} \quad \underline{\texttt{11}}\texttt{010}\underline{\texttt{11}} \quad \texttt{0101010}\underline{\texttt{00}}\texttt{10}\underline{\texttt{10}}\texttt{0}\underline{\texttt{00}}\texttt{10101} \quad \texttt{01}\underline{\texttt{000}}\texttt{101010101}.$$

Moreover, let $L_n = L \cap \bigcup_{i=0}^{n} \Sigma^i = \{s \in L : |s| \leq n\}$ be the subset of $L$ with strings of length $n$ or less.

**21.1)** For each of the following properties of a Turing machine $\mathcal{M}$, prove whether they are computable or not, invoking Rice's Theorem if possible:

$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L\} \\
\mathcal{P}_2 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L_5\} \\
\mathcal{P}_3 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L \text{ in less than } 10^6 \text{ steps}\} \\
\mathcal{P}_4 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L_5 \text{ in less than } 10^6 \text{ steps}\}
\end{aligned}$$

**21.2)** Write a single-tape, deterministic Turing machine (with alphabet $\{\textvisiblespace, \texttt{0}, \texttt{1}\}$) that decides $L$.

Hint — *Be careful in $\mathcal{P}_3$ and $\mathcal{P}_4$: we require $\mathcal{M}$ to halt within $10^6$ steps regardless of the input. The machine in 21.2 needs only perform a single scan, provided that it remembers two items of information: (i) the last symbol seen, and (ii) whether it has already met an exception (repeated digit) or not.*

**Solution 21**

**21.1)**

- $\mathcal{P}_1$ is non-trivial and semantic, hence it isn't computable.
  The property is clearly non-trivial because there are machines that decide $L$ —as proved by the fact that we are going to build one in 21.2— and machines that don't. The property is semantic because it only refers to the recognized language: suppose $L(\mathcal{M}_1) = L(\mathcal{M}_2)$; then either that language is $L$ (and hence $\mathcal{P}_1(\mathcal{M}1) = \mathcal{P}_1(\mathcal{M}_2) =$ true) or it isn't (in which case $\mathcal{P}_1(\mathcal{M}1) = \mathcal{P}_1(\mathcal{M}_2) =$ true). In both cases, $\mathcal{P}_1(\mathcal{M}1) = \mathcal{P}_1(\mathcal{M}_2)$.
  Hence, it satisfies the hypotheses of Rice's Theorem.

- $\mathcal{P}_2$ is non-trivial and semantic, hence it isn't computable.
  Same reasons as above. The very large upper bound to the number of steps guarantees that at least one machine has the property, hence it is non'trivial.

- $\mathcal{P}_3$ is trivial; this means that Rice's theorem isn't applicable, but also that the property is trivially computable.
  Indeed, $\mathcal{P}_3$ is trivially false; no machine can decide all infinite strings in $L$ within a fixed number of steps; if the input $x$ has more than $10^6$ symbols, no machine will ever be able to scan it within $10^6$ steps, so it cannot determine whether $x \in L$ or not.

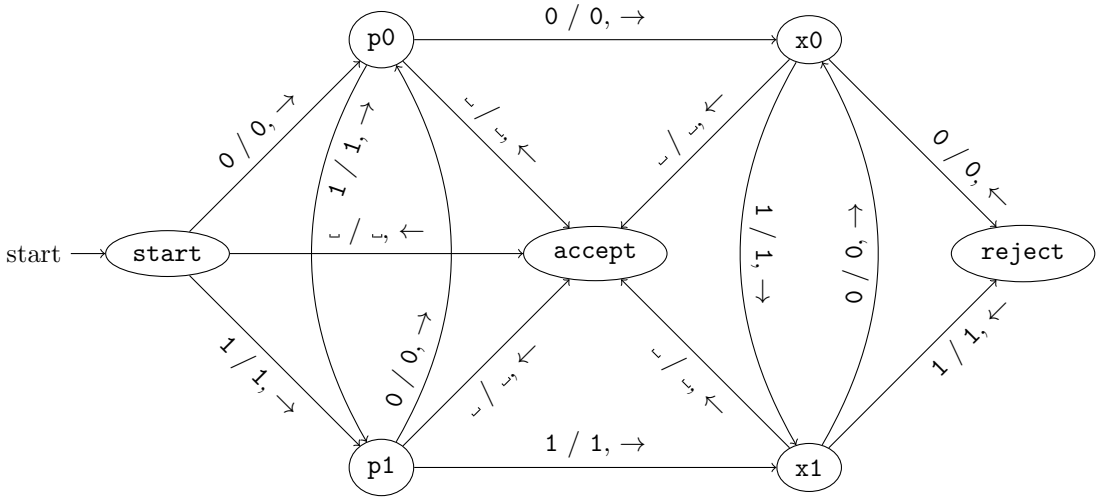
- $\mathcal{P}_4$ is non-trivial and non-semantic; Rice's theorem isn't applicable. It is computable, as shown below.
  It is neither trivial, nor semantic: we can take a machine $\mathcal{M}_1$ having property $\mathcal{P}_4$ and add a million dummy steps before halting. The machine still recognizes $L_5$, but it does not have property $\mathcal{P}_4$.

In order to test for $\mathcal{M} \in \mathcal{P}_4$, we just need to run $\mathcal{M}$ for at most $10^6$ steps on all possible combinations of symbols reachable in $10^6$ steps. Since all acceptable strings are much shorter (5 symbols at most), if the test succeeds we are sure that $\mathcal{M} \in \mathcal{P}_4$.

**21.2)** As suggested need to introduce separate states for all cases that the TM must remember; let's give them some mnemonic name:

- `p0`: previous symbol was a `0`, no exception found yet;
- `p1`: previous symbol was a `1`, no exception found yet;
- `x0`: previous symbol was a `0`, one exception found;
- `x1`: previous symbol was a `1`, one exception found;

None of the states listed above is eligible as initial, because in the beginning no previous symbol is memorized. The stateset above must thus be completed with a separate initial state (`start`); let us introduce two separate halting states for acceptance (`accept`) and rejection (`reject`).




**Observations**

- Why cannot we just apply to $\mathcal{P}_3$ the same reasoning we made about $\mathcal{P}_4$? Because $L$ also contains strings that are much longer than $10^6$ symbols, and if we just limit our tests to shorter strings we miss all of them. A procedure that only tests for strings up to a given size will accept a machine even if it fails on larger inputs (e.g., by rejecting strings that are longer that $10^6$ symbols, but still in $L$).
- It's important to keep a separate initial state, otherwise the machine will "remember" a non-existing previous symbol and discard too many strings.

**Exercise 22**

In the course we considered two versions of the halting problem: $\text{HALT}(\mathcal{M}, t)$, asking whether a machine $\mathcal{M}$ will halt on a specific input string $t \in \Sigma^*$, and $\text{HALT}_\varepsilon(\mathcal{M}) = \text{HALT}(\mathcal{M}, \varepsilon)$ on the empty input alone.
What about the two following languages?
**22.1)** asking whether a machine halts on all inputs:

$$\text{HALT}_\forall(\mathcal{M}) = \begin{cases} 1 & \text{if } \forall t \in \Sigma^* \; \mathcal{M}(t) < \infty \\ 0 & \text{otherwise.} \end{cases}$$

**22.2)** asking whether a machine halts on at least one input:

$$\text{HALT}_\exists(\mathcal{M}) = \begin{cases} 1 & \text{if } \exists t \in \Sigma^* \; \mathcal{M}(t) < \infty \\ 0 & \text{otherwise.} \end{cases}$$

Hint — *As you know, "halting" and "accepting" are quite interchangeable concepts; try to express the above decision functions as properties of languages accepted by the machine. But beware the direction of the reduction!*

**Solution 22**

**22.1)** Consider the property

$$\mathcal{P}_\forall = \{\mathcal{M} \,:\, L(\mathcal{M}) = \Sigma^*\} \quad \text{(i.e., machines that accept all strings).}$$

The property is clearly semantic and not trivial, therefore it isn't computable.
However, if $\text{HALT}_\forall$ were computable, then we could use it to compute $\mathcal{P}_\forall$: given a TM $\mathcal{M}$, modify it by replacing all rejection outcomes with infinite loops, and call $\mathcal{M}'$ the modified machine. It is clear that $\mathcal{M}$ accepts a string if and only if $\mathcal{M}'$ halts; therefore,

$$\mathcal{M} \in \mathcal{P}_\forall \quad \Leftrightarrow \quad \mathcal{M}' \in \text{HALT}_\forall.$$

We operated a Turing reduction from $\mathcal{P}_\forall$ to $\text{HALT}_\forall$. Since the former is not computable, the latter isn't either.
**22.2)** Same as above, but use the property

$$\mathcal{P}_\exists = \{\mathcal{M} \,:\, L(\mathcal{M}) \neq \emptyset\} \quad \text{(i.e., machines that accept at least one string).}$$

**Exercise 23**

For each of the following properties of Turing machines $\mathcal{M}$, prove whether it's computable or not. Invoke Rice's theorem whenever it applies.

$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} : \mathcal{M} \text{ accepts at least one input string}\} \\
\mathcal{P}_2 &= \{\mathcal{M} : \mathcal{M} \text{ accepts all input strings}\} \\
\mathcal{P}_3 &= \{\mathcal{M} : \mathcal{M} \text{ computes the Kolmogorov complexity of the input string}\} \\
\mathcal{P}_4 &= \{\mathcal{M} : \mathcal{M} \text{ halts on all input strings of length} \leq 10\} \\
\mathcal{P}_5 &= \{\mathcal{M} : \mathcal{M} \text{ halts in no more than 10 steps on all input strings}\}.
\end{aligned}$$

**Solution 23**

- $\mathcal{P}_1 = \{\mathcal{M} : \mathcal{M}$ accepts at least one input string$\}$ — **not computable**.
  The property refers to the accepted language: it can be rephrased as $L(\mathcal{M}) \neq \emptyset$, therefore it is semantic.
  More clearly, if two TMs accept the same language $L$, then either $L = \emptyset$ (thus neither machine has the property $cP_1$) or $L \neq \emptyset$ (and both machines have the property).
  The property is not trivial (there are machines that accept at least one string, and machines that do not accept any string). Therefore, Rice's theorem applies and the property is not recursive.
- $\mathcal{P}_2 = \{\mathcal{M} : \mathcal{M}$ accepts all input strings$\}$ — **not computable**.
  Again, the property is semantic (it can be rephrased as $L(\mathcal{M}) = \Sigma^*$) and not trivial (there are machines that accept all strings and machines that don't). Thus Rice's theorem applies.
- $\mathcal{P}_3 = \{\mathcal{M} : \mathcal{M}$ computes the Kolmogorov complexity of the input string$\}$ — **computable**.
  We know that Kolmogorov complexity is not computable; therefore the property is false for all machines. In other words, $\mathcal{P}_3 = \emptyset$ is trivially computed by any machine that rejects all inputs.
- $\mathcal{P}_4 = \{\mathcal{M} : \mathcal{M}$ halts on all input strings of length $\leq 10\}$ — **not computable**.
  We know that the halting problem is not computable even for inputs of length 0.
  More formally, we can prove that $\mathcal{P}_4$ is not computable in many ways:
  - By redefining the acceptance notion so that the act of halting implies acceptance (i.e., by saying that whenever a TM halts then the input is to be considered accepted). By doing this, $\mathcal{P}_4$ becomes semantic and non- trivial.
  - By reducing the $\text{HALT}_\varepsilon$ problem to it: given an arbitrary TM $\mathcal{M}$, to test $\mathcal{M} \in \text{HALT}_\varepsilon$, create $\mathcal{M}'$ that erases the input and then runs $\mathcal{M}$. Then, $\mathcal{M}$ halts on the empty input if and only if $\mathcal{M}'$ halts on every input (and in particular for all inputs of size up to 10):
    $$\mathcal{M} \in \text{HALT}_\varepsilon \Leftrightarrow \mathcal{M}' \in \mathcal{P}_4.$$
  - By identifying a TM that has property $\mathcal{P}_4$ (e.g., any machine that always halts) and mimicking the proof of Rice's Theorem.
- $\mathcal{P}_5 = \{\mathcal{M} : \mathcal{M}$ halts in no more than 10 steps on all input strings$\}$ — **computable**.
  The property is clearly not semantic (a machine halting in no more than 10 steps can easily be transformed into an equivalent machine without the property by adding 10 dummy states either on startup or before the halting state). So, Rice's theorem cannot be invoked.
  A simple method to verify the property is to simulate the execution of the $\mathcal{M}$ on every possible input and reject whenever a simulation lasts longer than 10 (simulated) steps. Although the number of input strings is infinite, the only relevant portion of the input is given by the tape cells reachable within 10 steps; therefore, the number of input strings to be tested is finite (at most 20 symbols, considering that TMs are bidirectional).

**Remarks**

- A short reason as to *why* each property is (or isn't) semantic or trivial would be appreciated.

**Exercise 24**

Let's consider a universal Turing machine $\mathcal{U}$ able to simulate 1-tape TMs on alphabet $\{0, 1, \textvisiblespace\}$.
Given a string $s \in \{0,1\}^*$, remember that its *Kolmogorov complexity* $K_{\mathcal{U}}(s)$ is the size of the smallest Turing machine (in $\mathcal{U}$'s notation) that, when started on an empty tape, outputs $s$ and halts.
Show that the set

$$\left\{s \in \{0,1\}^* | K_{\mathcal{U}}(s) \leq 10^6\right\}$$

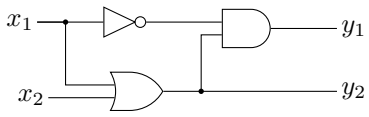
(binary strings having Kolmogorov complexity not larger than $10^6$) is recursively enumerable, by describing an algorithm to enumerate its elements.

**Solution 24**

The required set is the set of all the outputs of all machines of size not larger than $10^6$ that halt when executed on an initially empty tape.
First of all, since the set is obviously finite (the number of considered machines is finite, and each outputs at most one string), then it is implicitly recursive, in the sense that there must be a machine that decides it.
To prove that the set is RE in a constructive way, let us use the "procedural" definition of RE: a set is RE if there is a TM that prints all and only its elements.
To achieve this, we "just" need to simulate all machines of size not larger than $10^6$ "in parallel", i.e. by alternating a step of each machine. Every time a simulated machine halts, we take its tape and write it to some dedicated output tape. We cannot simulate them one after the other because some of them is never going to halt.
If a string $s \in \{0,1\}^*$ is such that $K_{\mathcal{U}}(s) \leq 10^6$, it means that one of the TMs that we are simulating will produce it as its output before halting, therefore it will eventually be written.
If, on the other hand, $K_{\mathcal{U}}(s) > 10^6$, then none of the simulated machines will ever produce $s$, and it will never be written out.

**Exercise 25**

Consider the following Boolean circuit:




**25.1)** Write down the Tseytin transformation of this circuit (i.e., the CNF formula that is satisfied by all and only combinations of input and output values compatible with the circuit).
**25.2)** Is it possible to assign input values to $x_1, x_2$ such that $y_1 = 0$ and $y_2 = 1$? Use the Tseytin transformation from 25.1 to provide a CNF formula that is satisfiable if and only if the answer is yes.
**25.3)** Is it possible to assign input values to $x_1, x_2$ such that $y_1 = 1$ and $y_2 = 0$? Use the Tseytin transformation from 25.1 to provide a CNF formula that is satisfiable if and only if the answer is yes.

**Exercise 26**

A *tautology* is a formula that is always true, no matter the truth assignment to its variables.
A Boolean formula is in *disjunctive normal form* (DNF) if it is written as the disjunction of clauses, where every clause is the conjunction of literals (i,e,, like CNF but exchanging the roles of connectives).
Let TAUTOLOGY be the language of DNF tautologies. Prove that TAUTOLOGY $\in$ **coNP**.

Hint — *You can do it directly (by applying any definition of **coNP**), or by observing that a tautology is the negation of an unsatisfiable formula, and that the negation of a CNF leads to a DNF.*

**Exercise 27**

What is the time complexity of your solution $\mathcal{M}$ to Exercise 2?
More precisely: if $n$ is the input size, what is the smallest exponent $k$ such that $\mathcal{M} \in \text{DTIME}(n^k)$?
Explain briefly.

**Exercise 28**

**28.1)** Let $L$ be a language on a finite alphabet, and let $\mathcal{N}$ be a non-deterministic Turing machine on the same alphabet with the following properties:

- $\mathcal{N}(x)$ takes at most $|x|^2$ non-deterministic steps before halting ($|x|$ is the size of $x$).
- If $x \notin L$, then $\mathcal{N}(x)$ rejects the input.
- If $x \in L$, then $\mathcal{N}(x)$ accepts the input.

Are these properties sufficient for us to say that $L \in$ **NP**?
**28.2)** Suppose that $\mathcal{N}$ has the following additional property:

- At every step, $\mathcal{N}$ performs at most one binary non-deterministic choice (i.e., $\mathcal{N}$ has two transition functions)

Given this property and those listed in the previous point, determine an upper bound for the number $C_{\mathcal{N}}(x)$ of non-deterministic computations performed by $\mathcal{N}$ on input $x$ as a function of the input size $|x|$.

**Exercise 29**

Let $L_1, L_2 \in$ **NP**. Does $L_1 \cup L_2 \in$ **NP**? Does $L_1 \cap L_2 \in$ **NP**? Why?

Hint — *Be as formal as you can, e.g.: "Since $L_1 \in$ **NP**, then there is a TM $\mathcal{M}_1$ such that..."*

**Exercise 30**

Consider the following classical **NP**-complete languages:

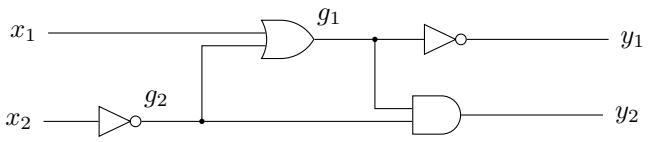
$$\begin{aligned} \text{CLIQUE} &= \{(G,k) : \text{Undirected graph } G \text{ has a completely connected subgraph of size } k\}, \\ \text{INDSET} &= \{(G,k) : \text{Undirected graph } G \text{ has a completely disconnected subgraph of size } k\}. \end{aligned}$$

**30.1)** Describe a polynomial-time reduction from one language to the other.
**30.2)** Show that $\text{CLIQUE} \cap \text{INDSET} \neq \emptyset$.

Hint — *For 30.1, choose the direction you like. In 30.2, don't be afraid of simple answers: to show that a set is not empty, you just need to find an element in it.*

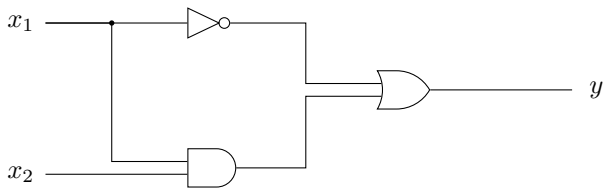
**Exercise 31**

Write the Tseytin transformation $f(x_1, x_2, y_1, y_2, g_1, g_2)$ that is satisfiable by precisely the truth values compatible with the following Boolean circuit:




**Exercise 32**

Consider the following Boolean circuit representing a Boolean function $y = f(x_1, x_2)$:




**32.1)** Write the function $f$ in terms of the Boolean operators $\wedge$ (and), $\vee$ (or) and $\neg$ (not) on the two variables $x_1$ and $x_2$.
**32.2)** Write a 3CNF formula on the three variables $x_1$, $x_2$ and $y$ (and, if needed, other auxiliary variables for gate outputs) that is satisfiable if and only if $y = f(x_1, x_2)$ (i.e., if $x_1$, $x_2$ and $y$ have values that are compatible with the given Boolean circuit).

Hint — *Point 32.2 can be solved in many ways: by directly writing the dependency as $y \Leftrightarrow f(x_1, x_2)$ and applying Boolean algebraic rules to work out a 3CNF formula, by writing down a 3CNF formula for each gate and requiring them all to be true, or by writing down the truth table and extracting the formula from there. The second way is the one discussed in the course.*

**Exercise 33**

Prove that the definition of recursive language does not change if we use non-deterministic TMs (i.e., every language that is recursive wrt DTMs is also recursive wrt NDTMs and vice versa).

**Exercise 34**

Show that INDEPENDENT SET $\leq_{\text{P}}$ INTEGER LINEAR PROGRAMMING by a direct reduction.

Hint — *Given a graph $G = (V, E)$, and an integer $k \in \mathbb{N}$, create an ILP problem with one variable for each vertex, constrained to $\{0, 1\}$, and try to come up with constraints that require the chosen vertices to be at least $k$ and not connected to each other.*

**Exercise 35**

Tweak the proof of Exercise 34 in order to reduce VERTEX COVER (in place of INDEPENDENT SET) to ILP.

Hint — *We need to transform the condition "every edge has at most one endpoint in the set", used in the aforementioned theorem, into the condition "every edge has* at least *one endpoint in the set"; the condition "there must be at least k 1's" must become "there must be* at most *k 1's".*

**Exercise 36**

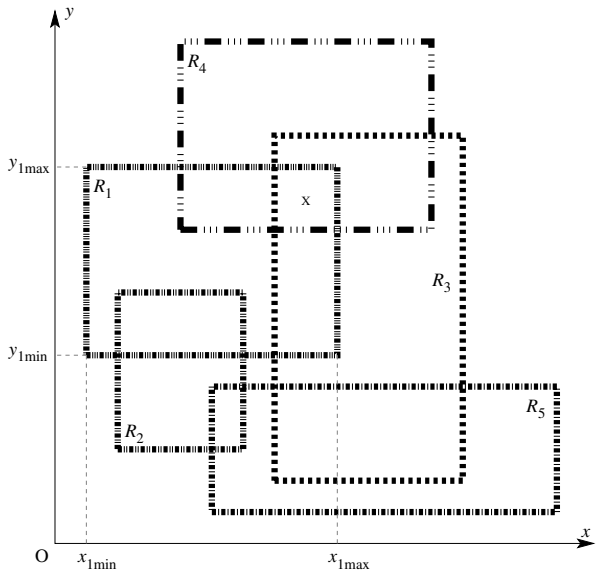
Consider the BOX DEPTH decision problem:

- let $n, k \in \mathbb{N}$;
- we are given $n$ rectangles $R_1, \ldots, R_n$ with sides parallel to the Cartesian axes; every rectangle $R_i$ is therefore defined by two integer $x$ coordinates and two integer $y$ coordinates:

$$R_i \equiv (x_{i\min}, x_{i\max}, y_{i\min}, y_{i\max}) \in \mathbb{N}^4, \qquad i = 1, \ldots n, \qquad x_{i\min} < x_{i\max}, \quad y_{i\min} < y_{i\max};$$

- we are asked if there is a region of the plane where at least $k$ rectangles overlap.

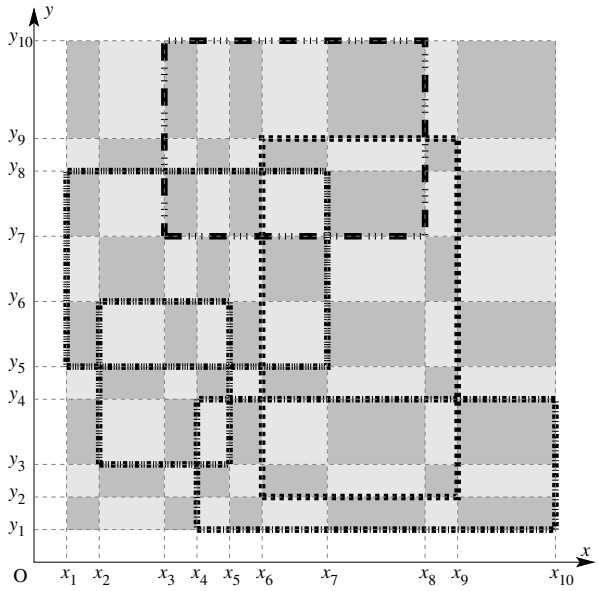
For instance, consider the following instance with $n = 5$ rectangles (the coordinates of $R_1$ are marked):




If $k \leq 3$, then the answer is "yes" (3 rectangles overlap in the region marked by "X"); if $k > 3$, then this instance has a negative answer.

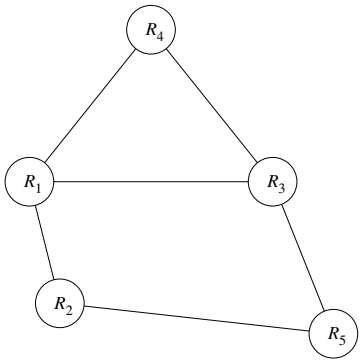
**36.1)** Prove that BOX DEPTH $\in$ **NP** by describing a suitable polynomial certificate for positive instances.

**36.2)** Observe that, if we mark all $x$ coordinates that define the rectangles on the $x$ axis and, likewise, all $y$ coordinates on the $y$ axis, we obtain a checkerboard like in the following figure, based on the previous 5-rectangle example:




Here the coordinates are listed as $x_1 \leq x_2 \leq \cdots \leq x_{2n}$ and $y_1 \leq y_2 \leq \cdots \leq y_{2n}$. Note that inside every cell $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$ $(i, j = 1, \ldots, n-1)$ of this checkerboard the number of overlapping rectangles doesn't change. Therefore, to solve the decision problem we just need to take one sample point for every cell and count the number of rectangles that contain it. Show that this technique takes polynomial time, therefore BOX DEPTH $\in \mathbf{P}$.

**36.3)** Note that there is an easy polynomial-time reduction from BOX DEPTH to CLIQUE: define a graph with a vertex for each rectangle, and join two vertices if the two corresponding rectangles overlap:




Observe that $k$ mutually overlapping rectangles correspond to a clique of size $k$, and there is a common overlap region. In our case, the triangle in the graph corresponds to the mutual overlap of $R_1$, $R_3$ and $R_4$. I.e., the box depth is $\geq k$ if and only if the graph has a clique of size $\geq k$.

So, BOX DEPTH, which has a polynomial-time algorithm, is reducible to CLIQUE, which is notoriously **NP**-complete. Have we just proved that $\mathbf{P} = \mathbf{NP}$?

**Exercise 37**

Consider the SET PACKING problem: given $n$ sets $S_1, \ldots, S_n$ and an integer $k \in \mathbb{N}$, are there $k$ sets $S_{i_1}, \ldots, S_{i_k}$ that are mutually disjoint?

**37.1)** Prove that SET PACKING $\in \mathbf{NP}$.

**37.2)** Prove that SET PACKING is **NP**-complete.

Hint — *You can prove the completeness by reduction of INDEPENDENT SET.*

**Exercise 38**

An examiner must plan an oral exam for $N$ students, where every student is asked one, and one only, question.
The examiner has the following information:

- a list of pairs of students who know each other (suppose that the relation is symmetric, but not transitive), and
- a number, $k$, of questions that she can ask.

We must determine whether the number of questions, $k$, is sufficient to avoid that two students knowing each other are asked the same question.
**38.1)** Describe a polynomial-time algorithm that decides the decision problem defined above when $k = 2$.
**38.2)** Prove that the problem is **NP**-complete in the general case (you can assume the **NP**-completeness of a language if it has been discussed in class).

**Exercise 39**

The police must intercept all cellphone communications within a group of $n$ people and have the following information:

- their identities and phone numbers (plus any other info that is needed in such cases);
- which pairs of people know each other (people who don't know each other will not directly communicate).

They want to know if there is a way to be sure to intercept all calls within group members while putting only $k$ phones under surveillance (we assume that a communication can be intercepted if at least one of the two phones is under surveillance).
**39.1)** Prove that the problem is **NP**.
**39.2)** Prove that the problem is **NP**-complete by reduction from some other known problem.

**Exercise 40**

Let $n$-SUBSET SUM be a restriction of SUBSET SUM to only instances of precisely $n$ numbers. The problem size can still be arbitrarily large, because the numbers may be as large as we want.
Would 1000000-SUBSET SUM be still in **NP**? Would it be **NP**-complete? Would it fall back to **P**?

**Exercise 41**

Let $M$-SUBSET SUM be a restriction of SUBSET SUM to only instances where all numbers $x_1, \ldots, x_n$ in the set (including the sum $s$) are not larger than $M$. The problem size can still be arbitrarily large, because the set can contain as many numbers as we want.
Would 1000000-SUBSET SUM be still in **NP**? Would it be **NP**-complete? Would it fall back to **P**?

**Exercise 42**

Consider the UNIVERSITY HIRING decision problem:

> A university needs to hire the teaching staff for a new degree, for which a set $T$ of topics must be taught. The executive board received $n$ applications from prospective teachers, and every applicant $i \in \{1, \ldots, n\}$ has knowledge of a subset $S_i \subseteq T$ of the required topics. The budget allows for hiring at most $k \leq n$ teachers. Is there a choice of $k$ applicants so that all teaching topics are covered?

An instance of the problem consists of parameters $n, k, T, S_1, \ldots, S_n$.
**42.1)** Prove that UNIVERSITY HIRING $\in$ **NP**.
**42.2)** Prove by reduction that UNIVERSITY HIRING is complete in the class **NP**. The reduction can refer to any language discussed during the course.
**42.3)** Prove that if $k$ is kept constant (e.g., $k = 10$), then the problem's asymptotic complexity is polynomial wrt input size.

**Exercise 43**

Remember that by "vertex coloring" of an undirected graph we mean the task of assigning an element from a finite set of labels ("colors") to each vertex of the graph so that no two connected vertices have the same color; we say that a graph can be $k$-colored if the task can be successfully carried out with no more than $k$ colors.
For each of the following coloring-related languages, determine whether they are in **P**, **NP** and/or **coNP** and provide a short motivation for your answers:

1. graphs that can be 2-colored;
2. graphs that cannot be 2-colored;
3. graphs that can be 3-colored;
4. graphs that cannot be 3-colored;
5. graphs $G = (V, E)$ that cannot be $k$-colored for any $k < |V|$.

Acceptable motivations can be formulated as short descriptions of algorithms (verbal or any kind of pseudocode), set inclusions (e.g., "I already proved that $L \in A$, but we know that $A \subset B$, therefore $L \in B$"), reductions to some **NP**-complete version of SATISFIABILITY (e.g., 3-SAT).

**Exercise 44**

A teacher wants to divide a large class of $N$ students into a small number $k \leq N$ of groups, each to be assigned a different project. She sets a constraint on how groups are formed:

***No-past-collaboration constraint**: Every group must be composed of students who never collaborated before — i.e., if two students already collaborated in a previous group project, then they must be placed in different groups.*

The teacher wants to know if she has prepared enough projects to be able to satisfy the constraint. Luckily for her, the College's Statistical Service maintains a comprehensive list of student groups formed in the past. Thus, she can formally define the following decision problem.

*Given (1) the number $N$ of students, (2) the list of past student collaborations (e.g., in the form of a list pairs of students) and (3) the number $k$ of available projects, is it possible to split the $N$ students into $k$ groups so that the **no-past-collaboration** constraint is satisfied?*
**44.1)** What complexity class does the above defined decision problem belong to, and why?
**44.2)** In particular, for what values of $k$ does an efficient decision procedure exist? Among them, for what values of $k$ is the decision trivial?
Note that group sizes need not be balanced, nor are we required to actually create the groups: we only need to decide if the number $k$ of projects is enough, or if the teacher needs to devise more of them.

**Exercise 45**

We know that the Traveling Salesman Problem (TSP) is **NP**-complete. Consider the following version:
**BOTTLENECK TSP** — *Given a complete, undirected graph $G = (V, E)$, with numeric costs associated to edges ($c : E \to \mathbb{N}$) and a budget $k$, is there a Hamiltonian cycle in $G$ where no traversed edges has a cost larger than $k$?*

In this variant, we don't bother about the overall cost, but only require that the most expensive traversed edge (the "bottleneck") doesn't cost more than our budget.
Show that BOTTLENECK TSP is **NP**-complete.

**Exercise 46**

In the central hall of the ancient temple, having dodged an inordinate amount of booby traps, Indiana Jones is ready to grab the golden statue from its pedestal; however, to avoid triggering even more deadly traps on his way back, he must replace the statue with something having its *exact* weight. Alas, his bag of sand was ripped by an arrow; all he has got is a large and heterogeneous set of archaeologist's tools whose individual weights he had luckily annotated in his notebook before leaving his office. Of course, his experienced look can precisely estimate the statue's weight he is so eager to match.

**46.1)** Show that, given the high precision of the ancient mechanism and the diversity of tools in his belt, he might need a very long[2] time before being able to determine if there is a combination of tools whose weight matches the statue's.

**46.2)** An oracle is quietly sitting in a corner of the hall. He was gifted with a very peculiar ability: when presented with any intricate map of rooms connected by tunnels, the oracle is immediately able to point out a round trip that visits all rooms exactly once, provided that such path exists. Prove that Indiana Jones could exploit the oracle's ability in order to solve the problem of matching the statue's weight in a reasonable[3] time.

Hint — *Assume that all weights are known with the precision of one gram. Both questions require answers in the form of reductions from/to known difficult (i.e., **NP**-complete) problems. Here is a list of languages that we already know to be **NP**-complete:* SATISFIABILITY, 3-SATISFIABILITY, CLIQUE, INDEPENDENT SET, INTEGER LINEAR PROGRAMMING, VERTEX COVER, 3-VERTEX COLORING, *SUBSET SUM*, KNAPSACK, HAMILTONIAN PATH, DIRECTED HAMILTONIAN CYCLE, *HAMILTONIAN CYCLE*, TRAVELING SALESMAN PROBLEM.

**Exercise 47**

The following example appears in the Clay Institute web page to motivate the inclusion the **P** vs. **NP** issue among its Millennium Prize Problems:

> *Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice.*

We are interested in the general problem with $N$ students and $n$ dormitory places (in the example, $N = 400$ and $n = 100$); let the question be "Will you be able to fill all $n$ dormitory places?"

**47.1)** Prove that the problem is in **NP**.

**47.2)** Knowing that 3SAT is complete, prove that the Clay Institute problem is **NP**-complete by an appropriate reduction.

Hint — *For point 47.2, observe that the Clay Institute example problem is just the rephrasing of a well known **NP**-complete problem, and imitate the reduction seen in class.*

**Exercise 48**

**48.1)** Prove that the two following problems belong to **NP**:

$P_1$: Given a finite list $L$ of unordered pairs of persons, where $\{a, b\} \in L$ means "$a$ and $b$ know each other", and a positive integer $k$, is there an individual who knows at least $k$ other people?

[2] I.e., exponential with respect to the number of tools in the worst case.
[3] I.e., polynomial with respect to the number of tools in the worst case.

$P_2$: Given a finite list $L$ of unordered pairs of persons, where $\{a, b\} \in L$ means "$a$ and $b$ know each other", and a positive integer $k$, is there a group of $k$ people who all know each other?

**48.2)** Prove the two following statements:

$$\text{If } P_2 \in \mathbf{P} \text{ then } \mathbf{P} = \mathbf{NP}.$$

$$\text{If } P_1 \text{ is } \mathbf{NP}\text{-complete then } P_2 \in \mathbf{P}.$$

Hint — *Here is a list of languages that you can assume to be **NP**-complete without having to prove it: SATISFIABILITY, 3-SATISFIABILITY, CLIQUE, INDEPENDENT SET, INTEGER LINEAR PROGRAMMING, VERTEX COVER, 3-VERTEX COLORING, SUBSET SUM, KNAPSACK, HAMILTONIAN PATH, DIRECTED HAMILTONIAN CYCLE, HAMILTONIAN CYCLE, TRAVELING SALESMAN PROBLEM.*

**Exercise 49**

In order to schedule a (very short) exam session, a University department must distribute $n$ exams among $m = 3$ time slots. In general $n$ can be very large, therefore many exams must take place at the same time.
To try to avoid conflicts, all students are asked to register to exams beforehand; two exams are *conflicting* if at least one student is registered to both. The department will try to assign conflicting exams to different time slots, so that no student has to sit through two exams at the same time. If such a schedule exists, we call it *non-conflicting*.
**49.1)** Prove that the problem of deciding the existence of a non-conflicting schedule is in **NP**. In particular, clarify what is the input and what is its size with respect to $n$ (remember that $m$ is fixed to 3).
**49.2)** Prove that, as the number $n$ of exams grows, the university policy is not scalable, in the sense that there is no known algorithm that can determine the existence of a non-conflicting exam schedule in polynomial time with respect to $n$ (again, remember that $m = 3$).
**49.3)** Provide polynomial algorithms to decide the problem when $m = 1$, $m = n - 1$ and $m \geq n$. Is the problem still polynomial wrt the number $n$ of exams when $m = 2$?

Hint — *Here is a list of known **NP**-complete problems for reference: SATISFIABILITY, 3-SATISFIABILITY, CLIQUE, INDEPENDENT SET, INTEGER LINEAR PROGRAMMING, VERTEX COVER, 3-VERTEX COLORING, SUBSET SUM, KNAPSACK, HAMILTONIAN PATH, DIRECTED HAMILTONIAN CYCLE, HAMILTONIAN CYCLE, TRAVELING SALESMAN PROBLEM.*

**Exercise 50**

Prove the following assertions by describing the appropriate polynomial-time reductions:
**50.1)** CLIQUE $\leq_{\mathrm{P}}$ INDEPENDENT SET;
**50.2)** INDEPENDENT SET $\leq_{\mathrm{P}}$ VERTEX COVER;
**50.3)** SATISFIABILITY $\leq_{\mathrm{P}}$ 3-SATISFIABILITY.

**Exercise 51**

The following statement is actually true, but what's wrong with the proof provided here?

**Theorem**: **NPSPACE** $\subseteq$ **PSPACE**

*Proof* — Let $L \in$ **NPSPACE**, let $\mathcal{N}$ be the NDTM that decides it in polynomial space, and let $x$ be an input string. Since we have no time bounds, we can emulate all possible computations of $\mathcal{N}(x)$, one after the other, until one of them accepts $x$ or we exhaust all of them. Of course, there is an exponential number of computations, but we have no time bounds, and each computation only requires polynomial space by definition: the tape can be reused between computations.
Therefore, we can emulate $\mathcal{N}$ by a deterministic, polynomial-space TM $\mathcal{M}$, and $L \in$ **PSPACE**, thus proving the assertion. □

**Exercise 52**

In a population of $n$ people, every individual only talks to individuals whom he knows. Steve needs to tell something to Tracy, but he doesn't know her directly.
We are provided with a (not necessarily symmetric) list of "who knows whom" in the group, and we are asked to tell whether Steve will be able to pass his information to Tracy via other people.
**52.1)** Is there a polynomial-time algorithm to give an answer (assume any realistic computing model you like)? If yes, describe it; if not (or if you cannot think of any), explain what is the main obstacle.
**52.2)** Is there a polynomial-space algorithm? Can we do any better? Describe the most space-efficient implementation that you can think of.

**Exercise 53**

A graph $G = (V, E)$ is connected if there is a path between every pair of nodes. Show that the language of connected graphs is in **NL**.

**Exercise 54**

As an embedded system programmer, you are asked to design an algorithm that solves SUBSET SUM on a (deterministic!) device with a $O(\log n)$ additional space constraint.
**54.1)** Should you start coding right away, should you argue that the solution is probably beyond your capabilities, or should you claim that the task is infeasible?
**54.2)** What space constraint would you be comfortable with, among $O(\log n)$, $O\big((\log n)^2\big)$, $O(n)$, $O(n^2)$, $O(2^n)$?

**Exercise 55**

Prove that the language $L$ defined in Exercise 2 belongs to the complexity class **L**.

**Exercise 56**

Consider the following language:

$$S = \Big\{ \big(x \in \{0,1\}^*, k \in \mathbb{N}\big) : x \text{ contains a subsequence of } k \text{ adjacent } 0\text{'s} \Big\}.$$

For example, $(101000101, 3) \in S$ because the binary string contains 3 consecutive zeroes, while $(101000101, 4) \notin S$ because the binary string does not contain 4 consecutive zeroes.
**56.1)** Prove that $S \in \mathbf{P}$.
**56.2)** Prove that $S \in \mathbf{L}$.

Hint — *Again, both points can be proved by describing an algorithm and showing that it has the required property.*

**Exercise 57**

Let $L$ be a **finite, non-empty** language on the alphabet $\Sigma = \{0, 1\}$.
For each of the following propositions say if it is true, false or (to the best of our knowledge) unknown, and briefly motivate your answer.

1. $L \in \mathbf{P}$.
2. $L \in \mathbf{L}$.
3. $L$ is polynomial-time reducible to 3SAT.
4. 3SAT is polynomial-time reducible to $L$.

**Exercise 58**

Let $L \in \mathbf{P}$ be a deterministic polynomial-time language on finite alphabet $\Sigma$, and let $L'$ and $L''$ be defined as follows:

- $L' = \Sigma^* \times L \times \Sigma^* = \{w_1 w_2 w_3 : w_1, w_3 \in \Sigma^* \wedge w_2 \in L\}$, the language of all strings on alphabet $\Sigma$ that contain a word from $L$ as a substring (contiguous sequence of symbols);
- $L'' = \{\sigma_1\sigma_2\sigma_3 \ldots \sigma_n \in \Sigma^* : \exists k, i_1, i_2, \ldots i_k (0 \leq k \leq n \wedge 1 \leq i_1 < i_2 < \cdots < i_k \leq n \wedge \sigma_{i_1}\sigma_{i_2} \ldots \sigma_{i_k} \in L)\}$, the language of all strings containing a (non necessarily contiguous) subsequence of symbols that compose a word in $L$.

For instance, if "cat" $\in L$, then "location" and "catalog" belong to both $L'$ and $L''$, while the words "decoration" and "croissant" only belong to $L''$.
**58.1)** Discuss the deterministic time complexity of $L'$ and $L''$.
**58.2)** What about their non-deterministic time complexity?

**Exercise 59**

The FACTORING decision problem is the following: given a pair of numbers $(n, k) \in \mathbb{N}^2$, does $n$ have a prime factor larger than or equal to $k$?
For instance, $(28, 5) \in$ FACTORING because 28 is divisible by 7, which is a prime number larger than 5.
On the other hand, $(27, 5) \notin$ FACTORING because the only prime that divides 27 is 3, which is less than 5.
**59.1)** Prove that FACTORING $\in$ **PSPACE**.

Hint — *This can be achieved in two ways.*
*You can prove it directly, by providing a simple algorithm that scans all prime numbers larger than $k$ and checks if any of them is a divisor of $n$, and showing that this algorithm is **PSPACE**.*
*Or you can prove it indirectly by showing that FACTORING belongs to a more convenient complexity class that is a subset of **PSPACE** (e.g., **NP**).*
***Bonus points** if you can give both proofs.*

**Exercise 60**

Consider the following language of strings in $\{\mathtt{a}, \mathtt{b}\}^*$:

$$L = \{\mathtt{a}^{n_1}\mathtt{b}\mathtt{a}^{n_2}\mathtt{b}\mathtt{a}^{n_3}\mathtt{b} \cdots \mathtt{b}\mathtt{a}^{n_k} : k \in \mathbb{N} \wedge k \geq 1 \wedge n_1 > n_2 > n_3 > \cdots > n_k > 0\}.$$

Namely, a string is in $L$ iff it is composed of non-empty sequences of a's of decreasing length separated by single b's, e.g.:

$$\mathtt{aaaaabaaaba} \in L,\ \mathtt{aaba} \in L,$$
$$\mathtt{aaabaaa} \notin L \text{ (two subsequences with the same number of a's: } n_1 = n_2 = 3),$$
$$\mathtt{aaabaaba} \in L,\ \mathtt{aaaaaaaaaaaaaaba} \in L,$$
$$\mathtt{aaabbaa} \notin L \text{ (two consecutive b's)},$$
$$\mathtt{aabaaa} \notin L \text{ (the sequences of b's are not of decreasing length)}.$$

Prove that $L \in \mathbf{L}$.

Hint — *Start by sketching down a (pseudocode) program that decides $L$.*

**Exercise 61**

Consider the language of all strings in $\{0, 1\}^*$where the number of 1's is strictly larger than the number of zeroes, e.g.:

$$011 \in L,\ 1101 \in L,$$
$$10 \notin L \text{ (the number of 1's must be } \textit{strictly} \text{ larger)},$$
$$1 \in L,\ 11111111 \in L,$$
$$0 \notin L,\ 101010 \notin L.$$

Prove that $L \in \mathbf{L}$.

**Exercise 62**

Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ be a three-symbol alphabet. Consider the language $L \subset \Sigma^*$ of strings where the three symbols have the same number of occurrences. For example:

$$\mathtt{abbcaccba} \in L \qquad \varepsilon \in L$$
$$\mathtt{abc} \in L \qquad \mathtt{aabb} \notin L$$
$$\mathtt{abacab} \notin L \qquad \mathtt{aaabbbccc} \in L$$

**62.1)** Prove that $L \in \mathbf{P}$.
**62.2)** Prove that $L \in \mathbf{L}$.

**Exercise 63**

Let $L_1, L_2 \in \mathbf{NP}$, and suppose $\mathbf{P} \neq \mathbf{NP}$. Does $L_1 \cup L_2 \in \mathbf{NP}$? Does $L_1 \setminus L_2 \in \mathbf{NP}$?
Try to be as clear and formal as possible.

**Exercise 64**

Consider the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and the language

$$L = \{\mathtt{a}^n\mathtt{bba}^n | n \in \mathbb{N}\} \subseteq \Sigma^*$$

i.e., a string belongs to $L$ if it contains zero or more `a`'s, followed by two `b`'s and then the same number of `a`'s, for example:

$$\mathtt{bb} \in L, \quad \mathtt{aabbaa} \in L, \quad \mathtt{aaabbaa} \notin L, \quad \mathtt{aaabaaa} \notin L.$$

**64.1)** Specify a one-tape, three-symbol deterministic Turing machine (on alphabet $\Sigma \cup \{\textvisiblespace\}$) that decides $L$.
What is the time complexity of your solution with respect to the input size (logarithmic, linear, quadratic, cubic, exponential...), and why?
**64.2)** Show that $L \in \mathbf{L}$.
**64.3)** Show that a two-tape DTM could decide $L$ in linear time.

Hint — *64.1: A DTM can be specified as a state-transition diagram or as a transition table indexed by symbols and states. If you cannot do it (or you have no time), then a verbal description is fine as long as it describes elementary actions of a TM (e.g., "count the number of `a`'s" isn't acceptable unless you specify how to do it). Remember that a one-tape DTM is allowed to modify its input (e.g., delete `a`'s as long as they are taken into consideration).*
*64.2 and 64.3: here a high-level description is fine, as long as it allows us to agree on the conclusion.*

**Exercise 65**

Let DOUBLE-SAT be the language of CNF Boolean formulas with *at least two* satisfying assignments.
**65.1)** Show that DOUBLE-SAT $\in \mathbf{NP}$.
**65.2)** Show that DOUBLE-SAT is **NP**-complete.

Hint — *For 65.2, try to reduce SAT to DOUBLE-SAT. Given CNF formula $f(x_1, \ldots, x_n)$, add a dummy variable $y$ to it obtaining $f'(x_1, \ldots, x_n, y)$ such that $f'$ is satisfiable if and only if $f$ is too, but $y$ can be either true or false, so that $f'$ has twice the number of solutions as $f$.*

**Exercise 66**

An astronomer has a list of 50 stars that must be photographed by an automated telescope during one night. The telescope starts from its initial resting position and must subsequently point to each of the 50 stars. The time for moving the telescope from one star to the next is proportional to their angular distance. After pointing (and shooting) to all the stars in the list, the telescope must move back to its resting position.
In order to minimize the expensive telescope time, the astronomer writes a program that, given the list of star coordinates, computes the optimal photographing sequence, i.e., the sequence of telescope movements starting and ending at the resting position, covering all 50 stars, and taking the shortest possible time. The program does not directly drive the telescope, and can be run at any moment during the day: on the 50-star list it takes 10 minutes and outputs the optimal sequence. In the night, the sequence is fed to the actual telescope driver and is completed in 2 hours of telescope time.
A few days later, the astronomer has a new list of twice as many (100) stars to be photographed on the next night. In the morning they fire up their program, but it keeps crunching numbers for the whole day until the astronomer, out of patience, kills it.
**66.1)** Why did this happen? Identify a known hard problem and show by reduction that this behavior is, indeed, expected.
**66.2)** Using the existing program with a reasonably short runtime, is it still possible for the astronomer to plan a (possibly suboptimal) sequence that still takes all 100 required photographs in one night?

Hint — *For 66.2, you might want to consider a* divide et impera *strategy.*
*For simplicity, assume that all required stars are visible throughout the night.*

**Exercise 67**

**67.1)** Is the HALT language **NP**-hard?
**67.2)** Is HALT **NP**-complete?

Hint — *Motivate your answer starting from the relevant definitions. While this question has a definitely correct answer, you are allowed to express well-motivated doubts!*

**Exercise 68**

An instance $(k, G)$ of the CLIQUE language (where $k$ is the required clique size and $G$ is an undirected graph) is encoded with alphabet $\Sigma = \{\text{␣}, \texttt{0}, \texttt{1}, \texttt{+}\}$ as follows:

- the binary encoding of $k$,
- followed by a blank cell,
- followed by the adjacency matrix of $G$, where every row is represented as a sequence of `0`'s and `1`'s, and rows are separated by a `+` symbol.

**68.1)** Does the instance encoded by the following string belong to the CLIQUE language or not?

`11␣0101+1010+0101+1010`

**68.2)** Write a Turing machine that performs a reduction from the INDEPENDENT SET to the CLIQUE language, using the same encoding for the instances $(k, G)$ of both languages. What is the time complexity of the reduction?

# Appendix C

# Old exercises

The following exercises are not suitable for the latest edition of the course because they require topics from Chapter 5. They are all marked as such in order to avoid confusion.

**Exercise 69**

**NB** — not suitable for the 2024-25 edition of the course.
Show that **P** $\subseteq$ **ZPP**.

Hint — *A polynomial-time language is automatically in **ZPP** because...*

**Exercise 70**

**NB** — not suitable for the 2024-25 edition of the course.
Show that **RP** $\subseteq$ **BPP**.

Hint — *The condition for a language to be in **RP** can be seen as a further restriction on those imposed on **BPP**.*

**Exercise 71**

**NB** — not suitable for the 2024-25 edition of the course.
Show that **BPP** $\subseteq$ **PP**.

Hint — *The conditions for a language to belong to a class automatically satisfy those for the other.*

**Exercise 72**

**NB** — not suitable for the 2024-25 edition of the course.
Given the following formulation of the decision problem IDEPENDENT SET (INDSET):

0. Given the graph $G = (V, E)$ and $k \in \mathbb{N}$, is there an independent set in $G$ of size at least $k$?

Consider the following functional versions:

1. Given the graph $G = (V, E)$, return the size $k$ of the largest independent set.
2. Given the graph $G = (V, E)$, return an independent set of maximum size.
3. Given the graph $G = (V, E)$ and $k \in \mathbb{N}$, return an independent set of size at least $k$.

For which indices $i, j = 0, \ldots, 3$ are the following propositions true/false/unknown?

1. Version $i$ immediately reduces to version $j$ (i.e., an answer to $j$ immediately provides an answer to $i$).
2. If we had an oracle for version $j$, we could answer version $i$ by repeated calls to $j$.

3. Version $i$ is in **NP**.

4. Version $i$ is in **FNP**.

**Exercise 73**

**NB** — not suitable for the 2024-25 edition of the course.
Let $L$ be a language, and let $\mathcal{N}$ be a non-deterministic Turing Machine that decides $x \in L$ in time $O(|x|^3 \log |x|)$.
**73.1)** Suppose that, whenever $x \in L$, at least 15 computations of $\mathcal{N}(x)$ accept; what probabilistic complexity classes does $L$ belong to, and why?
**73.2)** Suppose that, whenever $x \in L$, at most 15 computations of $\mathcal{N}(x)$ do not accept; what probabilistic complexity classes would $L$ belong to, and why?

Hint — *Consider the following classes:* ***RP***, ***coRP***, ***ZPP***, ***BPP***, ***PP***. *Bonus points if you also consider* ***P*** *and* ***NP***.

**Exercise 74**

**NB** — not suitable for the 2025-26 edition of the course.
We know that Post's Correspondence Problem (PCP) is not recursive, because every computation of a Turing machine is reducible to an instance of PCP.
Consider the following version of PCP:
**BOUNDED PCP** — *Given $n$ string pairs $(a_i, b_i)$, $i = 1, \ldots, n$, is there a sequence of possibly repeated indexes $i_1, i_2, \ldots, i_k = 1, \ldots, n$, with $k \leq n$, such that $a_{i_1} a_{i_2} \ldots a_{i_k} = b_{i_1} b_{i_2} \ldots b_{i_k}$?*
The definition only adds a bound to the number of selected string pairs, requiring that they are no more than $n$, while the original version did not set any bound to the solution length.
**74.1)** Show that this bounded version of the problem is computable by describing a simple algorithm to solve it; discuss the time complexity of your algorithm (arguably exponential).
**74.2)** Prove that BOUNDED PCP $\in$ **NP** by describing a suitable certificate for positive instances of the problem.

# List of lectures

This list provides links to video recordings and approximate page references for lectures in the Fall 2026 season.
The beginning and end paragraphs of a lecture are also marked in the text margins.


Monday, 2026-09-14 ..... video and slides
Tuesday, 2026-09-15 ..... 7–10, video
Monday, 2026-09-21 ..... 10–11, video
Tuesday, 2026-09-22 ..... 12–13, video
Monday, 2026-09-28 ..... 13–14, video
Tuesday, 2026-09-29 ..... 14–18, video
Monday, 2026-10-05 ..... 18–20, video
Tuesday, 2026-10-06 ..... 21–21, video
Monday, 2026-10-12 ..... 21–23, video
Tuesday, 2026-10-13 ..... 23–25, video
Monday, 2026-10-19 ..... Exercises, video
Tuesday, 2026-10-20 ..... 26–28, video
Monday, 2026-10-26 ..... 31–32, video
Tuesday, 2026-10-27 ..... 32–36, video
Monday, 2026-11-02 ..... 36–37, video
Tuesday, 2026-11-03 ..... 37–39, video
Monday, 2026-11-09 ..... 39–43, video
Tuesday, 2026-11-10 ..... 43–46, video
Monday, 2026-11-16 ..... 47–51, video

# Index


3-SAT, *see* $k$-SATISFIABILITY

accept, 21
alphabet, 7

Berry paradox, 30
Boolean circuit, 43
Busy Beaver, 22

certificate, *see* **NP** certificate
Church-Turing thesis, 14
CNF, 35
co-R.E. set, *see* co-recursively enumerable set
co-recursively enumerable set, 20
Collatz
    conjecture, 72
    sequence, 72
commutative property
    of Boolean operators, 35
computable
    function, 21
    language, 15
computation
    as graph of configurations, 47, 48
compute, 21
**coNEXP**, 63
configuration, 47
    encoding of, 48
conjunctive normal form, *see* CNF
**coNP**, 38
Cook-Levin Theorem, 50
countable set, 8

De Morgan's laws, 35
decidable
    language, 15
decide, 21
decision function, 7
diagonal argument, 10, 73
DIRECTED HAMILTONIAN CYCLE, 59
disjunctive normal form, *see* DNF
distributive property
    of Boolean operators, 35
DNF, 38

DSPACE($f$), *see* space class, deterministic
DTIME($f$), *see* time class, deterministic

empty string, 7
eventually outgrow, 22
existential-mode NDTM, 39
**EXP**, 33, 63
exponential-time language, *see* **EXP**, *see* **EXP**

FACTORING, 61

HALT, *see* Halting problem
$\text{HALT}_\varepsilon$, *see* Halting problem, with empty input
Halting problem, 16
    with empty input, 17
Hamiltonian
    cycle, 59
    path, 56
HAMILTONIAN CYCLE, 59
HAMILTONIAN PATH, 56

ILP, *see* INTEGER LINEAR PROGRAMMING
INTEGER LINEAR PROGRAMMING, 41

$k$-CNF, 40
$k$-SAT, *see* $k$-SATISFIABILITY
$k$-SATISFIABILITY, 40
$k$-VERTEX COLORING, 51
Kleene closure, 7
KNAPSACK, 55
Kolmogorov complexity, 26

**L**, 65
$\mathbf{L}^2$, 70
language, 10
logarithmic-space language, *see* **L**
    non-deterministic, *see* **NL**

NDTM, *see* Turing machine, non-deterministic
**NEXP**, 63
**NL**, 65
**NP**, 36
**NP** certificate, 36
**NP** witness, *see* **NP** certificate

**NP**-complete language, 50
**NP**-hard language, 50
NSPACE($f$), *see* space class, non-deterministic
NTIME($f$), *see* time class, non-deterministic

**P**, 33
polynomial-time language, *see* **P**
    Non-deterministic, *see* **NP**
    quasi-, *see* **QP**
property
    of a Turing machine, 24
    semantic, 24
    trivial, 24

**QP**, 70
quasi-polynomial time language, *see* **QP**

R.E. set, *see* recursively enumerable set
REACHABILITY, *see* STCON
recognize, *see* accept
recursive
    language, 15
recursively enumerable set, 20
reduction
    polynomial-time, 39
    Turing, 17
RESTRICTED HALT, 63
Rice's theorem, 24

SAT, *see* SATISFIABILITY
SATISFIABILITY, 32
Savitch's Theorem, 69
SET COVER, 53
snapshot, *see* configuration
space class
    deterministic, 65
    non-deterministic, 65
ST-CONNECTIVITY, *see* STCON
STCON, 66
SUBSET SUM, 53

TAUTOLOGY, 38
time class
    deterministic, 32
    non-deterministic, 37
TM, *see* Turing machine
transition function, 11
TRAVELING SALESMAN PROBLEM, 55, 61
truth table, 43
Tseytin transformation, 45
TSP, *see* TRAVELING SALESMAN PROBLEM
Turing machine, 11
    non-deterministic, 37
    simulators, 12
    universal, 14
    with 2-symbol alphabet, 13
    with multiple tapes, 12

universal-mode NDTM, 39
UTM, *see* Turing machine, universal

VERTEX COVER, 53

witness, *see* **NP** certificate