Computability and Computational Complexity
Academic year 2018–2019, first semester
Lecture notes

Mauro Brunato

Version: 2018-12-14

Warning

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

`https://comp3.eu/`

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

Changelog

2018-12-14

- Various typos, corrected `path_exists` function in proof of Theorem 25 to return Boolean values instead of halting (courtesy of Elia Grego).

2018-12-13

- More **NP**-complete problems (SUBSET SUM, KNAPSACK)
- The Merkle-Hellman Knapsack cryptosystem
- Zero-knowledge proofs
- Final remarks
- Exercises

2018-12-10

- Interactive proof systems

2018-12-04

- Space complexity classes.
- Function problems

2018-11-26

- Errata: consistent use of c as **NP** certificate, definition of ILP (thanks to Elia Grego).
- Probabilistic complexity classes.
- Questions and exercises.

2018-11-16

- Changed reducibility sign from $<_P$ to $\leq_p$.
- Added definitions and proofs on Boolean circuits, **NP**-hardness and completeness.
- Added parts on **EXP**, **NEXP** and **RP**.
- added questions and exercises about **NP**-complete problems and Boolean circuits.

2018-11-05

- Computational complexity: introductory part (**P**, **NP**, reductions).

2018-10-10

- Answers to exercises.

2018-10-08

- Rice's theorem, non-computability of the BB function.

2018-10-05

- Added some nomenclature and exercises.

2018-10-01

- Fourth and fifth lecture: existence of uncomputable functions, the halting problem, co-R.E. functions and sets, the Busy Beaver game.

2018-09-24

- Third lecture: alphabet size, Universal Turing machines, emulation of other computational systems.

2018-09-19

- Second lecture: definition of Turing machine, examples, equivalence of multiple-tape machines.

2018-09-14

- First lecture: basic definitions, Collatz example

Contents

Part I

Lecture notes

Chapter 1

Computability

1.1 Basic definitions and examples

In computer science, every problem instance can be represented by a finite sequence of symbols from a finite alphabet, or equivalently as a natural number. Therefore, we can restrict our focus on functions mapping natural numbers to natural numbers:

$$f : \mathbb{N} \to \mathbb{N}.$$

In particular, we are interested in functions for which the answer is "yes" or "no", which can be modeled as

$$f : \mathbb{N} \to \{0, 1\};$$

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

Examples:

- Given a natural number n, is n prime?
- Given a graph, what is the maximum degree of its nodes?
- From a customer database, select the customers that are more than fifty years old.
- Given a set of pieces of furniture and a set of trucks, can we accommodate all the furniture in the trucks?

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

Decision functions and sets

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

$$S_f = \{n \in \mathbb{N} : f(n) = 1\}.$$

Conversely, given a natural number subset $S \subseteq \mathbb{N}$, we can always define the function that decides over elements of the set:

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

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

Examples of computable functions and sets are the following:

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

1.1.1 A possibly non-recursive set

Collatz numbers

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

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

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

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

Suppose that we are asked the following question:

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

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

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

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

Decidability of the Collatz property

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

In more formal terms, we are admitting that we are **not** able to prove that the Collatz property is *decidable* (i.e., that there is a computer program that always terminates with the correct answer[4]). However, we have provided a procedure that terminates with the correct answer when the answer is

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

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

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

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

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

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

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

"yes" (the function is not *total*, in the sense that it doesn't always provide an answer). We call such set **recursively enumerable**[5] (or RE, in short).

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

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

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

1.2 A computational model: the Turing machine

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

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

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

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

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

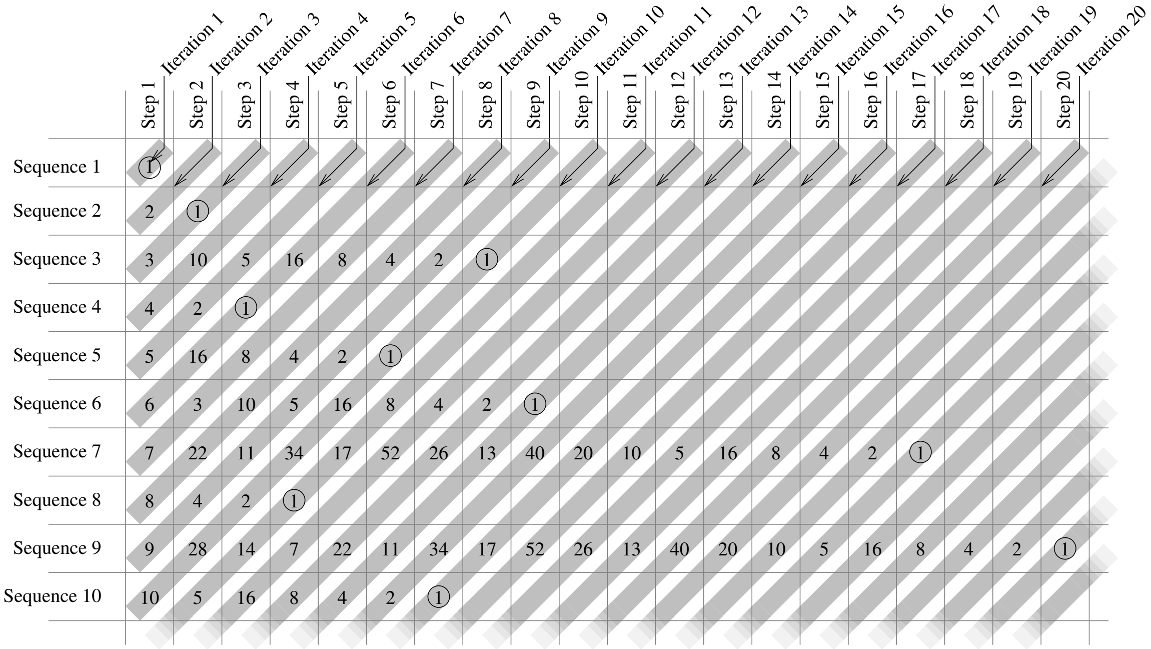
The machine writes the symbol σ' on the current tape cell, enters state q', and moves the current position by one cell in direction d. If the machine enters one of the distinguished halting states, then

[5] https://en.wikipedia.org/wiki/Recursively_enumerable_set

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

[7] https://en.wikipedia.org/wiki/Turing_machine

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



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

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

the computation ends. At this point, the content of the (non-blank portion of) the tape represents the computation's *output*.

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

1.2.1 Examples

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

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

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

1.2.2 Computational power of the Turing Machine

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

Multiple-tape Turing machines

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

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

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

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

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

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

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

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

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

Size of the alphabet

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

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

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

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

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

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

□

1.2.3 Universal Turing machines

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

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

One such machine, using a 16 symbol encoding and a single tape, is described in `https://www.dropbox.com/sh/u7jsxm232giwown/AADTRNqjKBIe_QZGyicoZWjYa/utm.pdf`

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

Simulating other computational devices

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

1.2.4 The Church-Turing thesis

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

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

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

1.3 Uncomputable functions

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

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

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

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

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

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

We know that strings can be enumerated: first we count the only string in Σ^0, then the strings in Σ^1, then those in Σ^2 (e.g., in lexicographic order), and so on. Since every string $s \in \Sigma^*$ is finite

($s \in \Sigma^{|s|}$), sooner or later it will be enumerated. Therefore there is a mapping $\mathbb{N} \to \Sigma^*$, i.e., Σ^* is countable.

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

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

Corollary 1. *There are uncomputable decision functions.*

1.3.1 Finding an uncomputable function

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

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

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

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

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

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

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

is uncomputable.

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

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

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

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

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

Proof. Let's proceed by contradiction. Suppose that we have a machine $\mathcal{H}$ which computes $HALT(s,t)$ (i.e., when run on a tape containing a string s encoding a TM and a string t, always halts returning 1 if machine $\mathcal{M}_s$ would halt when run on input t, and returning 0 otherwise). Then we could use $\mathcal{H}$ to compute function UC.

For convenience, let us compute UC using a machine with two tapes. The first tape is read-only and contains the input string $\alpha \in \Sigma^*$, while the second will be used as a work (and output) tape. To compute UC, the machine will perform the following steps:

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

This machine would be able to compute UC by simply applying its definition, but we know that UC is not computable by a TM; all steps, apart from $\mathcal{H}$, are already known and computable. We must conclude that $\mathcal{H}$ cannot exist.

See also Arora-Barak, theorem 1.17 in the public draft. □

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

The Haliting Problem for machines without an input

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

Let us define the following specialized halting function:

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

It turna out that if we weer able to compute $HALT'$ then we could also compute $HALT$:

Theorem 5. *$HALT'$ is not computable.*

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

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

Consequences of the Halting Problem incomputability

If $HALT$ were computable, we would be able to settle any mathematical question that can be disproved by a counterexample (on a discrete set), such as the Collatz conjecture, Goldbach's conjecture[8], the non-existence of odd perfect numbers[9]... We would just need to write a machine that systematically search for one such counterexample and halts as soon as it finds one: by feeding this machine as an input to $\mathcal{H}$, we would know whether a counterexample exists at all or not.

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

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

1.3.2 Recursive enumerability of halting computations

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

It is also possible to output all $(s,t) \in \Sigma^* \times \Sigma^*$ pairs for which $\mathcal{M}_s(t)$ halts by employing a diagonal method similar to the one used in Fig. 1.2[10].

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

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

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

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

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

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

[9] https://en.wikipedia.org/wiki/Perfect_number

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

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

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

1.3.3 Another uncomputable function: the Busy Beaver game

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

Definition 2 (The Busy Beaver game). *Among all TMs on alphabet $\{0, 1\}$ and with $n = |Q|$ states (not counting the halting one) that halt when run on an empty (i.e., all-zero) tape:*

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

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

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

Theorem 7. *The function $S(n)$ is not computable.*

Proof. Suppose that $S(n)$ is computable. Then, we could create a TM to compute $HALT'$ (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
```

□

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 8. *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:

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

By definition, F clearly has the following properties:

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

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

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

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

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

- Write x ones on the tape and stop at the rightmost one (i.e., the unary representation of x: it can be done with x states, see Exercise 3 at page 62);

- Execute M_F on the tape (therefore computing $F(x)$ with C states);

- Execute M_F again on the tape (therefore computing $F(F(x))$ with C more states).

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

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

Now,

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

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

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

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

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

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

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

□

This proof is based on the original one by Rado (1962)[11].

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

1.3.4 More definitions

A few more definitions, for brevity:

- Given an alphabet Σ, a set of strings $S \subseteq \Sigma^*$ is also called a **language**.
- 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.

1.3.5 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 σ on the tape?"...

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

Definition 3. *A* property *of a TM is a mapping P from TMs to $\{0,1\}$, and we say that $\mathcal{M}$ has property P when $P(M) = 1$.*

Definition 4. *A property is* semantic *if its value is shared by all TMs recognizing the same language: if $L(\mathcal{M}) = L(\mathcal{M}')$, then $P(\mathcal{M}) = P(\mathcal{M}')$.*

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

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

Theorem 9 (Rice's Theorem)**.** *All non-trivial semantic properties of TMs are undecidable.*

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

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

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

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

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

Chapter 2

Complexity classes: P and NP

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

2.1 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

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

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:

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

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

2.2 Polynomial languages

Let us now focus on decision problems.

Definition 5. *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 $L \in \mathit{DTIME}(f)$, if there is a TM $\mathcal{M}$ that decides L and its worst-case time, as a function of input size, is dominated by f:*

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

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 $L \in \text{DTIME}(f)$ means that there is a machine $\mathcal{M}$, a constant $c \in \mathbb{N}$ and an input size $n_0 \in \mathbb{N}$ such that, for every input x with size larger than n_0, $\mathcal{M}$ decides $x \in L$ in at most $c \cdot f(|x|)$ steps.

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

Definition 6.

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

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

$$x \in L \quad \Leftrightarrow \quad \mathcal{M}(x) = 1 \wedge t_{\mathcal{M}}(x) \le p(|x|). \tag{2.1}$$

2.2.1 Examples

Here are some examples of polynomial-time languages.

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

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

(Counter?)-examples

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

SATISFIABILITY or SAT — (see also Sec. 2.4.1) Given a Boolean expression $f(x_1, \dots, x_n)$ (usually in conjunctive normal form, CNF[3]) involving n variables, is there a truth assignment to the variables that satisfies (i.e., makes true) the formula[4]?

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

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

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

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

INDEPENDENT SET or INDSET — Given an encoding of graph G and a number k, does G contain k nodes that are not connected to each other[5]?

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

$$\left(\sum_{i=1}^{n-1} w_{\sigma_i\sigma_{i+1}}\right) + w_{\sigma_n\sigma_1} \leq k, \tag{2.2}$$

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

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

2.3 NP languages

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

- If $f(x_1, \dots, x_n) \in$ SAT (i.e., boolean formula f is satisfiable), then there is a truth assignment to the variables $x_1, \dots, x_n$ that satisfies it. If we were given this truth assignment, we could easily check that, indeed, $f \in$ SAT. Note that the truth assignment consists of n truth values (bits) and is therefore shorter than the encoding of f (which contains a whole boolean expression on n variables), and that computing a Boolean formula can be reduced to a finite number of scans.
- If $G \in$ INDSET, then there is a list of k independent nodes; given that list, we could easily verify that G does not contain any edge between them. The list contains k integers from 1 to the number of nodes in G (which is polynomial with respect to the size of G's representation) and requires a presumably quadratic or cubic time to be checked.
- If $G \in$ TSP, then there is a permutation of the nodes in G, i.e., a list of nodes. Given that list, we can easily sum the weights as in (2.2) and check that the inequality holds.

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

Definition 7. *We say that a language $L \subseteq \Sigma^*$ is of class $\boldsymbol{NP}$, and write $L \in \boldsymbol{NP}$, if there is a TM $\mathcal{M}$ and two polynomials $p(n)$ and $q(n)$ such that for every input string x*

$$x \in L \quad \Leftrightarrow \quad \exists c \in \Sigma^{q(|x|)} : \mathcal{M}(x, c) = 1 \land t_{\mathcal{M}}(x, c) \leq p(|x|). \tag{2.3}$$

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.

Theorem 10.

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

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

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

Proof. The first inclusion, $\mathbf{P} \subseteq \mathbf{NP}$, is trivial if we consider that definition 7 reduces to definition 6 if we set $q(n) = 0$, i.e., we accept an empty certificate: languages in P are polynomially verifiable without need of certificates.
For the second inclusion, to decide $L \in \mathbf{NP}$ we just need a machine that iterates through all certificates $c \in \Sigma^{q(|x|)}$ and runs every time $\mathcal{M}(x, c)$, accepting x as soon as a pair (x, c) is accepted or rejecting x when all certificates are exhausted. This can be done in some adequately large time 2^{n^k}. □

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

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

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

We can define the class NTIME(f) as the NDTM equivalent of class DTIME(f), just by replacing the TM in Definition 5 with a NDTM:

Definition 9. *Let $f : \mathbb{N} \to \mathbb{N}$ be any computable function. We say that a language $L \subseteq \Sigma^*$ is of class NTIME(f), and write $L \in$ NTIME(f), if there is a NDTM $\mathcal{M}$ that decides L and its worst-case time, as a function of input size, is dominated by f:*

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

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

Theorem 11.

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

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

Let $L \in \mathbf{NP}$, as in Definition 7. 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 7 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

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

$x \in L$, 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 7, 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 7 hold, and $L \in \mathbf{NP}$. □

2.4 Reductions and hardness

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

Definition 10. *Given two languages $L, L' \in$ **NP**, we say that L is* polynomially reducible *to L', and we write $L \leq_p L'$, if there is a function $R : \Sigma^* \to \Sigma^*$ such that*

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

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

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

2.4.1 Example: Boolean formulas and the conjunctive normal form

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

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

Definition 11 (Conjunctive normal form)**.** *A formula f is said to be in* conjunctive normal form *with n variables and m clauses if it can be written as*

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

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

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

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) \quad = \quad & (x_1 \vee \neg x_2 \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \\ & \wedge (\neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5). \end{aligned} \qquad (2.4)$$

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 (2.4):

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

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

Definition 12 (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, (2.4) 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 13. *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 12. *Given $k \in \mathbb{N}$,*

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

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

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 13.

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

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

$$\bigvee_{j=1}^{l_i} g_{ij} \tag{2.5}$$

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

$$\left(h \vee \bigvee_{j=1}^{l_i-2} g_{ij} \right) \wedge (\neg h \vee g_{i,l_i-1} \vee g_{il_i}), \tag{2.6}$$

by keeping all literals, apart from the last two, in the first clause, and putting the last two in the second one. By construction, the truth assignments that satisfy (2.5) also satisfy (2.6), and viceversa. In fact, if (2.5) is satisfied then at least one of its literals are true; but then one of the two clauses of (2.6) is satisfied by the same literal, while the other can be satisfied by appropriately setting the value of the new variable h. Conversely, if both clauses in (2.6) are satisfied, then at least one of the literals in (2.5) is true, because the truth value of h alone cannot satisfy both clauses.

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

As an example, the 4-CNF formula (2.4) can be reduced to the following 3-CNF with the two additional variables h and k used to split its two 4-clauses:

$$\begin{aligned} f'(x_1, x_2, x_3, x_4, x_5, h, k) \quad = \quad & (h \vee x_1 \vee \neg x_2) \wedge (\neg h \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \\ & \wedge (k \vee \neg x_1 \vee \neg x_2) \wedge (\neg k \vee x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5). \end{aligned} \tag{2.7}$$

Theorem 13 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 12, 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 $\in$ **P**.

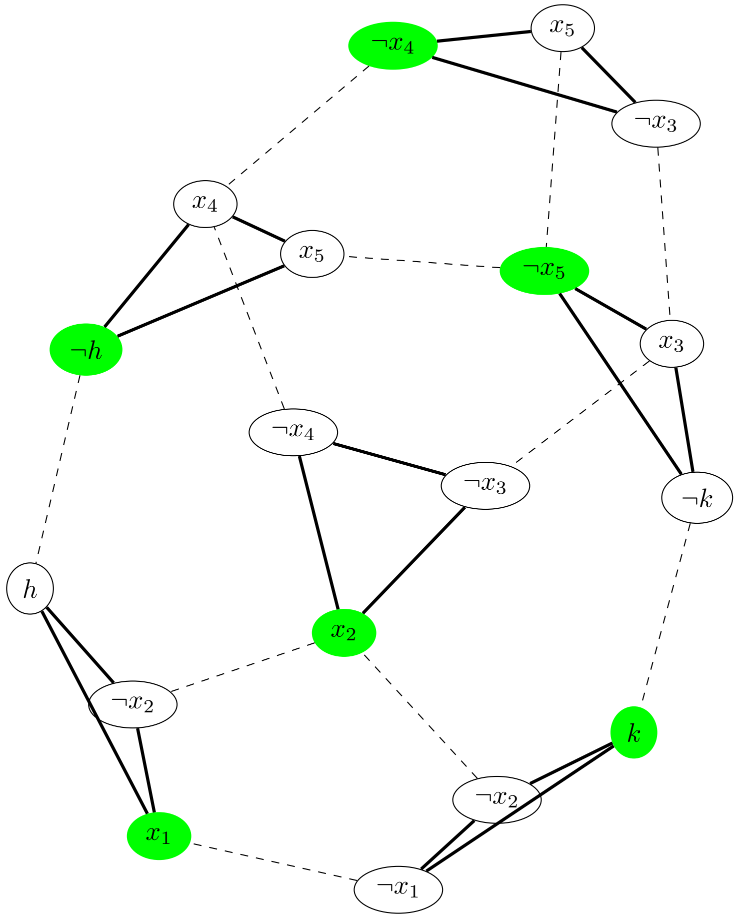


Figure 2.1: Reduction of the 3-CNF formula (2.7) to a graph for INDSET.

2.4.2 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 14.

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

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. 2.1 shows the graph corresponding to the 3-CNF formula (2.7): each bold-edged triangle corresponds to one of the six clauses, with every node labeled with one of the literals. The dashed edges connect every literal with its negations.

It is easy to see that the original 3-CNF formula is satisfiable if and only if the graph contains an independent set of size $k = m$ (number of clauses). Given the structure of the graph, no more than one node per triangle can appear in the independent set (nodes in the same triangle are not independent), and if a literal appears in the independent set, then its negation does not (they would be connected by an edge, thus not independent). If the independent set has size m, then we are ensured that one literal per clause can be made true without contradictions. As an example, the six green nodes in Fig. 2.1 form an independent set and correspond to a truth assignment that satisfies f. □

2.5 NP-hard and NP-complete languages

Definition 14. *A language L is said to be* **NP**-hard *if for every language $L' \in$ **NP** we have that $L' \leq_p L$.*

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

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

Definition 15. *A language $L \in$ **NP** that is **NP**-hard is said to be* **NP**-complete*.*

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

2.5.1 CNF and Boolean circuits

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

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

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

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

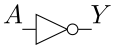
$$\begin{aligned} a \vee (b \wedge c) &\equiv (a \vee b) \wedge (a \vee c) \\ a \wedge (b \vee c) &\equiv (a \wedge b) \vee (a \wedge c) \\ \neg(a \vee b) &\equiv \neg a \wedge \neg b \\ \neg(a \wedge b) &\equiv \neg a \vee \neg b \\ a \rightarrow b &\equiv \neg a \vee b. \end{aligned}$$

Another convenient 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. 2.2 shows the gates corresponding to the fundamental Boolean connectives, together with their truth tables and CNF formulae.

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

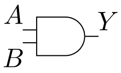
Any combinational Boolean circuit can be "translated" into a CNF formula, in the sense that the formula is satisfiable by all and only the combinations of truth values that satisfy the circuit. Given a Boolean circuit with n inputs $x_1, \dots, x_n$ and m outputs $y_1, \dots, y_m$ and l gates $G_1, \dots,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

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



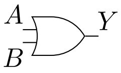
A	Y
1	0
0	1

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



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

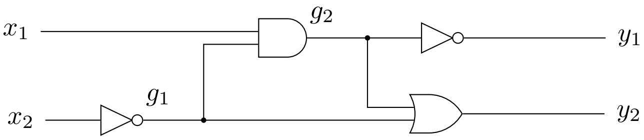
$$\begin{aligned} & Y = A \wedge B \\ \equiv\ & \big(\neg Y \vee (A \wedge B)\big) \wedge \big(Y \vee \neg(A \wedge B)\big) \\ \equiv\ & (\neg Y \vee A) \wedge (\neg Y \vee B) \wedge (Y \vee \neg A \vee \neg B) \end{aligned}$$



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

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

Figure 2.2: A NOT gate (top), an AND gate (middle) and an OR gate (bottom), their truth tables, and derivations of the CNF formulae that are satisfied if and only if their variables are in the correct relation (i.e., only by combinations of truth values shown in the corresponding table).



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

Figure 2.3: A Boolean circuit and its CNF representation: the CNF formula is satisfiable by precisely the combinations of truth values that are compatible with the logic gates.

Fig. 2.3 shows an example: a Boolean circuit with 2 inputs, 2 outputs and 2 ancillary variables associated to intermediate gates, together with the corresponding CNF formula. This formula completely expresses the dependency between all variables in the circuit, 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 SAT by replacing $y_1 = y_2 = 0$ (and, of course, $\neg y_1 = \neg y_2 = 1$) in the CNF of Fig. 2.3, 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 $g_2 \wedge \neg g_2$.

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.

2.5.2 Using Boolean circuits to express Turing Machine computations

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

	0	1
s_1	$1, s_1, \rightarrow$	$1, s_2, \leftarrow$
s_2	$0, s_1, \leftarrow$	$0, \mathtt{HALT}, \rightarrow$

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

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

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

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

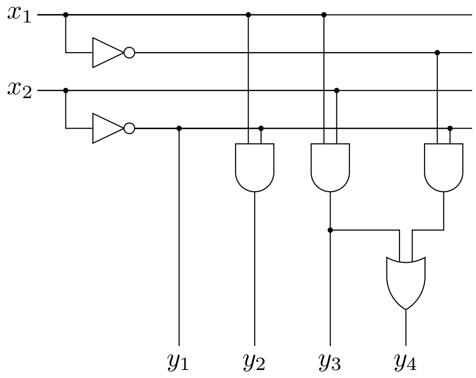


Figure 2.4: The Boolean circuit that implements the transition table of the TM described in the text.

Fig. 2.4 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.

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 3. *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 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. 2.4, 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. 2.5), 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 4. *Lemma 3 also works if the TM is non-deterministic.*

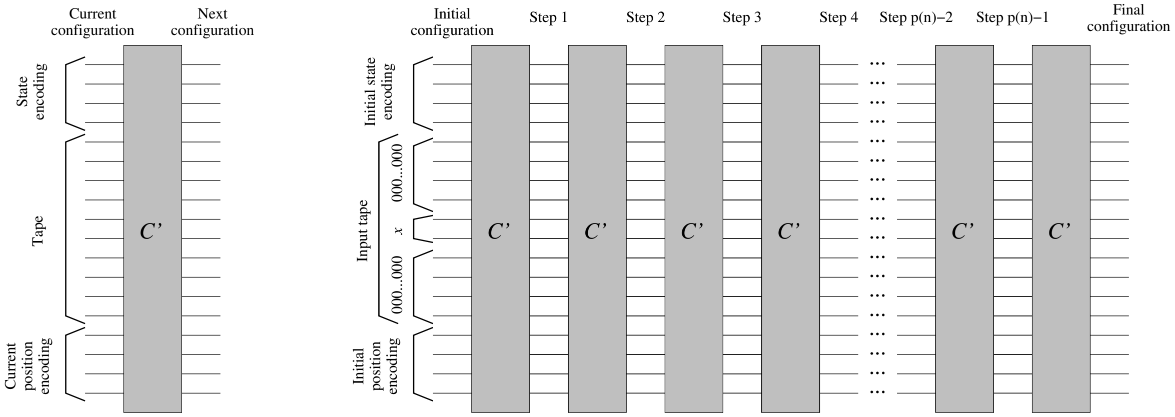


Figure 2.5: (left) C' is a Boolean circuit with a polynomial number of inputs, gates and outputs with respect to the size of the TM's input x. It transforms a Boolean representation of a configuration of the TM into the configuration of the subsequent step. (right) By concatenating $p(|x|)$ copies of C', we get a polynomial representation of the whole computation of the TM on input x.

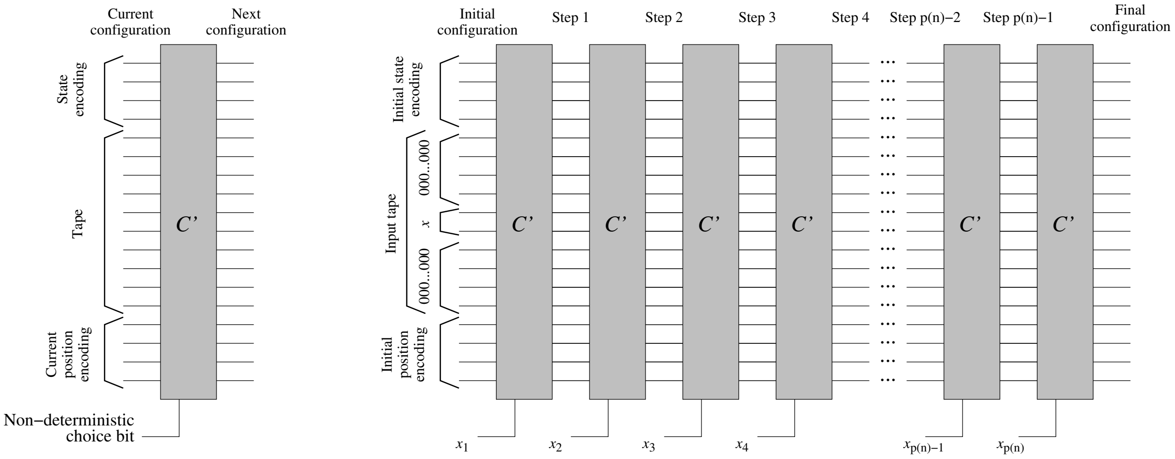


Figure 2.6: Analogous to Fig. 2.5 for a NDTM. (left) Every C' block has an additional input that allows the selection of the non-deterministic choice for the step that it controls. (right) The whole circuit has $p(n)$ additional Boolean inputs $x_1, \ldots, x_{p(n)}$: every combination of choice bits represents one of the $2^{p(n)}$ computations of the NDTM.

Proof outline. See Fig. 2.6: in order to carry on a NDTM's computation, we just need to modify the circuit C' of Lemma 3 to accept one further input, and use it to choose between the two possible outcomes of the transition table. Let us call $x_1, \dots, x_{p(n)}$ the additional input bits of the daisy-chained C' blocks. Each of the $2^{p(n)}$ combinations of these inputs determines one of the possible compuattions of the NDTM. □

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.

Theorem 15 (Cook's Theorem)**.** *SAT is* ***NP****-hard.*

Proof outline. To prove this, we need to pick a generic language $L \in \mathbf{NP}$ and show that $L \leq_p \text{SAT}$.

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 4, we can build a Boolean circuit C with polynomial size that, for any truth value combination of the inputs $x_1, \dots, x_{p(n)}$, 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. 2.5.1.

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 $x_1, \dots, x_{p(n)}$.

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,

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

□

Of course, we already know that SAT $\in$ **NP**, hence the following:

Corollary 2. *SAT is* ***NP****-complete.*

2.6 Other 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:

Lemma 5. *If L is* ***NP****-hard and $L \leq_p L'$, then also 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 13, and from the fact that 3-SAT $\in$ **NP**, we get:

Lemma 6. *3-SAT is **NP**-complete.*

Next, from Theorem 14, and from the fact that INDSET $\in$ **NP**, we get:

Lemma 7. *INDSET is **NP**-complete.*

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

CLIQUE — Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, does G have a complete subgraph (a "clique") of size (at least) k?

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'?

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 $x_1, \dots, x_n \in \mathbb{Z}$?

VERTEX COLORING — Given an undirected graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, is there an assignment from V to $\{1, \dots, k\}$ ("k colors") such that two connected vertices have different colors?

SET COVER — Given a finite set W, n subsets $S_1, \dots, S_n$ and an integer $k \in \mathbb{N}$, are there k subsets $S_{i_1}, S_{i_2}, \dots, S_{i_k}$ such that their union is W?

It is easy to see that all such languages are in **NP**.

Observe that a clique of size k in $G = (V, E)$ corresponds to an independent set of the same size in the complementary graph $\bar{G} = (V, \bar{E})$ (where $\bar{E}$ is the complement of E), and viceversa, therefore complementing E is a polynomial-time reduction from INDSET to CLIQUE:

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

Also, observe that if V' of size k is an independent set in $G = (V, E)$, then its complement $V \setminus V'$ is an independent set of size $|V| - k$ and viceversa. Therefore:

Lemma 9. *VERTEX COVER is **NP**-complete.*

Here follow a few slightly more complex reductions.

Theorem 16. *SET COVER is **NP**-complete.*

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

We start from VERTEX COVER. Given a graph $G = (V, E)$, let $W = E$ in the SET COVER definition, and map every vertex $i \in V$ to set

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

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 W (every edge of G) belongs to at least one of them (has an endpoint in one of these vertices). □

Theorem 17. *ILP is **NP**-complete.*

Proof. First, ILP is clearly in **NP**.

We start from VERTEX COVER. Given a graph $G = (V, E)$, and an integer $k \in \mathbb{N}$, we can set up some constraints such that we create an integer program whose solutions imply a k-vertex cover for G and vice versa.

Let's create an integer program vith one variable per vertex in V. We want these variables to encode the inclusion of a vertex in the cover ($x_i = 1$ if vertex i is in G's cover, 0 otherwise). Since in INTEGER PROGRAMMING all variables can be arbitrary integers, we restrict them between 0 and 1 by setting the inequalities $-x_i \leq 0$ and $x_i \leq 1$ for $i = 1, \dots, |V|$. The requirement that $x_1, \dots, x_{|V|}$ is a cover is implemented by introducing a constraint for every edge $i, j \in E$ that requires at least one of the endpoints to be 1: $x_i + x_i \geq 1$. Finally, the requirement that the size of the cover is at most k is encoded in $x_1 + x_2 + \cdots + x_{|V|} \leq k$.

In conclusion, the following integer program has a solution if and only if the corresponding graph has a cover of size k:

$$\begin{cases} - & x_i & & & & & \leq & 0 & \qquad \forall i \in V \\ & x_i & & & & & \leq & 1 & \qquad \forall i \in V \\ - & x_i & - & x_j & & & \leq & -1 & \qquad \forall \{i, j\} \in E \\ & x_1 & + & \dots & + & x_{|V|} & \leq & k & \end{cases}$$

□

Theorem 18. *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. 2.7.

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. □

Other examples can be found in Karp's seminal paper from 1972[8] in which he lists 21 **NP**-complete problems. The interested reader is invited to have a look at the paper and check some of his reductions.

A special mention goes to the GRAPH ISOMORPHISM language: given two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are they isomorphic (e.g., is there a bijection $f : V_1 \to V_2$ such that $f(E_1) = E_2$)? Obviously, GRAPH ISOMORPHISM $\in$ **NP**: the bijection, if it exists, can be checked

[8] Richard M. Karp, *Reducibility among Combinatorial Problems*, 1972
`http://cgi.di.uoa.gr/~sgk/teaching/grad/handouts/karp.pdf`

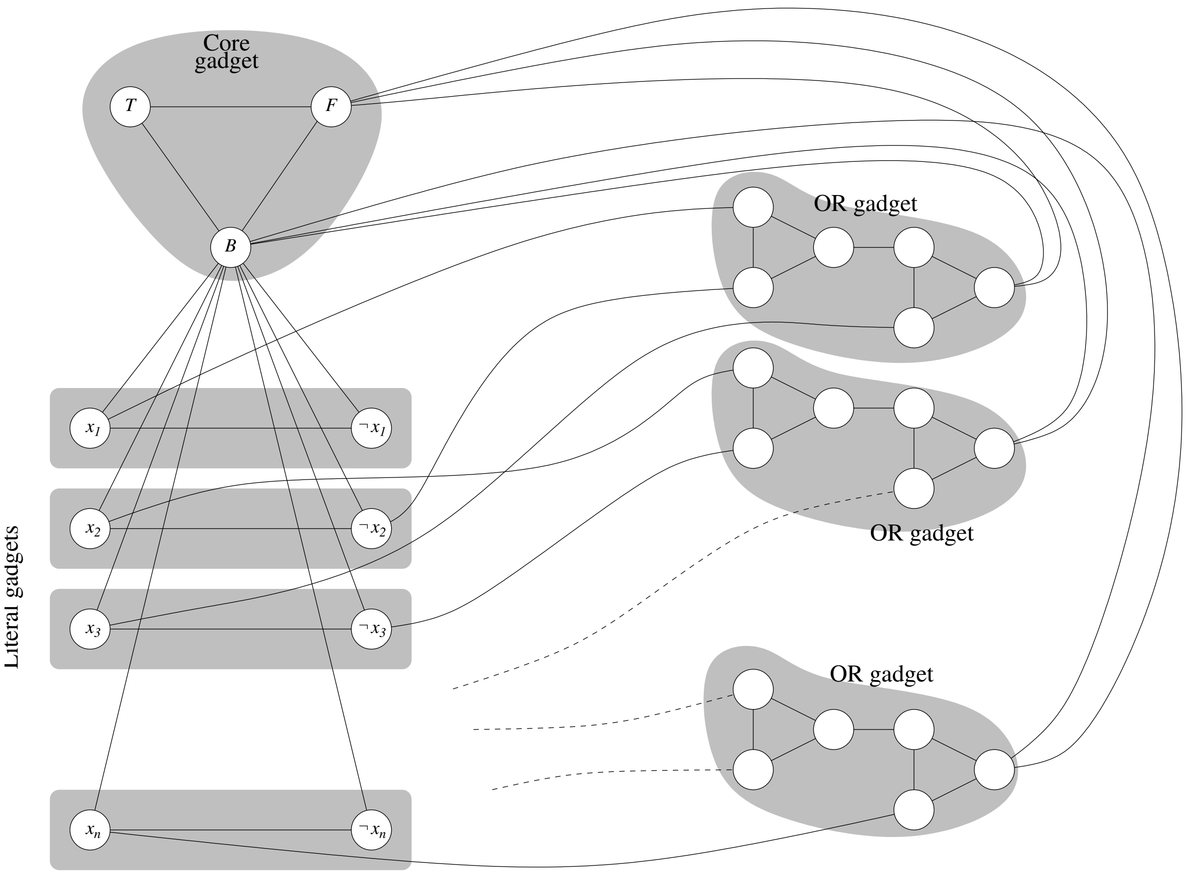


Figure 2.7: Reduction of a 3-CNF formula to the VERTEX COLORING problem with $k = 3$ colors.

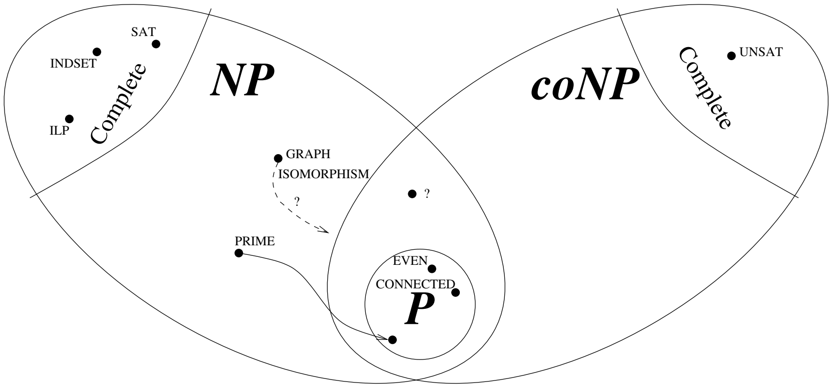


Figure 2.8: What we know up to now. If any of the **NP** or **coNP**-complete problems were to be proven in **P**, then all sets would collapse into it.

in polynomial time. However, it is believed that the language is not complete. In fact, there is a *quasi-polynomial*[9] algorithm that decides it.

2.7 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 16. *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.*

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

Clearly, $\mathbf{P} \in \mathbf{NP} \cap \mathbf{coNP}$ because in **P** everything is polynomially verifiable. Currently, we don't know if the inclusion is strict or not.

Fig. 2.8 summarizes what has been said in this chapter.

[9] more than polynomial, but less than exponential, e.g., $\mathrm{DTIME}(2^{c_1(\log n)^{c_2}})$

Chapter 3

Other complexity classes

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

3.1 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 17.

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

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.

Cominq 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 10.

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

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. □

An important result is that the analysis of the relationship between **EXP** and **NEXP** can help wrt the **P** vs. **NP** problem:

Theorem 19. *If* $\boldsymbol{EXP} \neq \boldsymbol{NEXP}$*, then* $\boldsymbol{P} \neq \boldsymbol{NP}$*.*

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 $2^{|x|^c}$.

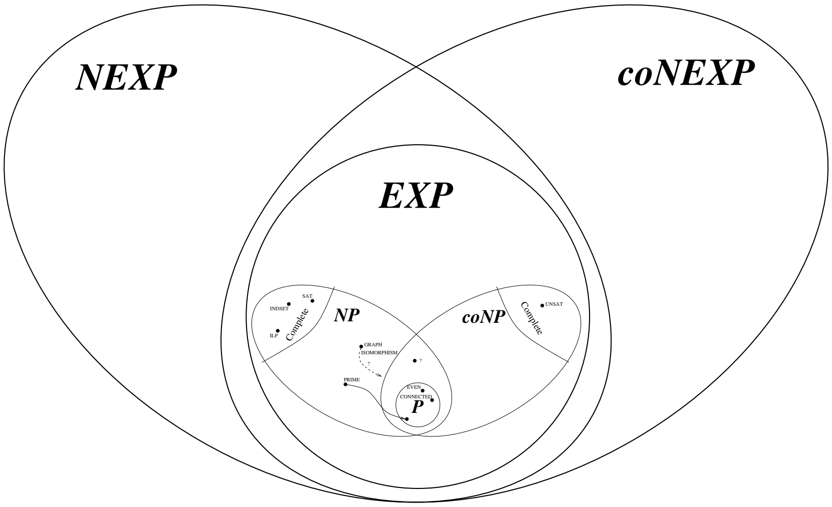


Figure 3.1: The exponential classes. The inner part is shown in greater detail in Fig. 2.8.

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 $L \in \mathbf{EXP}$. □

Fig. 3.1 summarizes the addition of the exponential classes.

3.2 Randomized complexity classes

Observe that the definition of **NP** only requires one computation out of an exponentially large number 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).

3.2.1 The classes RP and coRP

Ley us define the following complexity class:

Definition 18. *Let $L \in$ **NP**, and let $0 < \varepsilon < 1$. We say that L is* randomized polynomial time*, and write $L \in$ **RP**, if there is a NDTM $\mathcal{M}$ that accepts L in polynomial time and, whenever $x \in L$,*

$$\frac{\textit{Number of accepting computations of } \mathcal{M}(x)}{\textit{Number of computations of } \mathcal{M}(x)} \geq \varepsilon. \tag{3.1}$$

Obviously, if $x \notin L$ then there are no accepting computations. In other words, if $L \in$ **RP** we are guaranteed that, whenever $x \in L$, a sizable number of computations accept it[1].

Theorem 20.

$$\boldsymbol{P} \subseteq \boldsymbol{RP} \subseteq \boldsymbol{NP}.$$

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 (3.1) 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 $L \in$ **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, $\Pr\bigl(\mathcal{M}(x,c) = 1\bigr) \geq \varepsilon$. Conversely, if $x \notin L$ then $\Pr\bigl(\mathcal{M}(x,c) = 1\bigr) = 0$, 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 ε, 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 ε of computations reject x if $x \notin L$.

[1] See the Arora-Barak draft, Chapter 7, until Section 7.1 included, for definitions based on "probabilistic Turing Machines"

Examples

There are very few "natural" examples of languages in **RP** (or **coRP**) that do not belong to **P**. What constituted the main example, PRIMES, is now proved to be in **P**.

Definition 19. *Let $P = \mathbb{Z}/p\mathbb{Z}[x_1, \dots, x_n]$ 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.:*

$$f(x_1, \dots, x_n) = (2x_1 + x_2 - 3x_4 + 2x_6 - 1) \cdot (5x_2 + 4x_3 + x_6 + 2) \cdots (x_2 + 4x_5 + x_{n-1} - 3x_n - 5). \quad (3.2)$$

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 $x_1, \dots, x_n$) and verifying that all coefficients are zero. However, transforming the form (3.2) into the canonical form would require an exponential number of multiplications and sums.

The algorithms to decide PIT rely[2] 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 ε, which is precisely the definition of **coRP**.

3.2.2 Zero error probability: the class ZPP

An interesting characterization of **RP** and **coRP** is the following:

- $L \in$ **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 $1 - \varepsilon$;

- $L \in$ **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 $L \in$ **RP** $\cap$ **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

$$\forall x \in \Sigma^* \quad \forall c \in \{0,1\}^{p(|x|)} \qquad \Pr\bigl(M_1(x,c) \text{ accepts}\bigr) \text{ is } \begin{cases} 0 & \text{if } x \notin L \\ \geq \varepsilon_1 & \text{if } x \in L \end{cases}$$

and

$$\forall x \in \Sigma^* \quad \forall c \in \{0,1\}^{p(|x|)} \qquad \Pr\bigl(M_2(x,c) \text{ rejects}\bigr) \text{ is } \begin{cases} 0 & \text{if } x \in L \\ \leq \varepsilon_2 & \text{if } x \notin L. \end{cases}$$

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
```

[2] See https://en.wikipedia.org/wiki/Polynomial_identity_testing and https://en.wikipedia.org/wiki/Schwartz%E2%80%93Zippel_lemma if interested; the PIT problem is also described in Arora-Barak (draft), Section 7.2.2.

Observe that this algorithm does not define an explicit number of iterations. However, if $x \in L$, at every iteration M_1 has probability ε_1 to accept it, after which the algorithm would stop; conversely, if $x \notin L$, at every iteration M_2 has probability ε_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 20. ***ZPP*** $=$ ***RP*** $\cap$ ***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 21.

$$\boldsymbol{P} \subseteq \boldsymbol{ZPP} \subseteq \boldsymbol{RP} \subseteq \boldsymbol{NP}.$$

Proof. The proof is left as an exercise (exercise 8). □

3.2.3 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 21. *A language L is said to be* bounded-error probabilistic polynomial*, written $L \in$* ***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
```

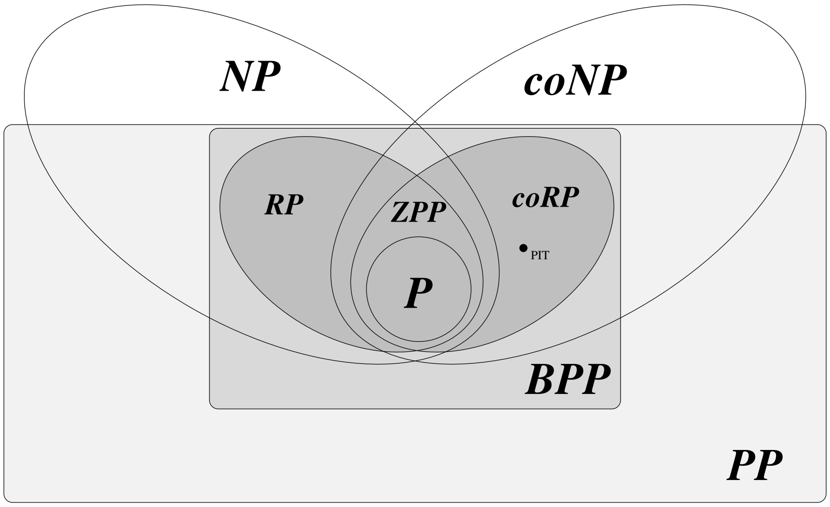


Figure 3.2: What we know about the probabilistic classes introduced in this Section. In particular, the relationship between **BPP** and **NP** is unknown.

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 22. *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, rether it could be exponential wrt $|x|$).

Given the above definitions, the following theorem should be obvious:

Theorem 22.

$$\boldsymbol{RP} \in \boldsymbol{BPP} \subseteq \boldsymbol{PP}.$$

Proof. The proof is left as an exercise (exercises 9 and 10). □

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 **NP** $\subseteq$ **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.

Fig. 3.2 summarizes what has been said in this Section.

3.3 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 23. *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 24. *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).*

3.3.1 Logarithmic space classes: L and NL

Definition 25.

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

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

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

is the same if non-deterministic compuattions are allowed.

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 23.

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

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 26. *A triplet composed of a* directed *graph $G = (V, E)$ and two nodes $s, t \in V$ belongs to the CONNECTIVITY (or ST-CONNECTIVITY, or STCON) 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:

```
on input $G = (V, E)$; $s, t \in V$
    current $\leftarrow s$                                   "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 $\leftarrow$ a node adjacent to current       among all adjacent nodes
    reject and halt
```

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 24.

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

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:

Theorem 25.

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

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

```
on input $G = (V, E)$; $s, t \in V$
    path_exists $\leftarrow$ function ($v$, $w$, $l$)
        if $v = w \vee \big(l = 1 \wedge (v, w) \in E\big)$
            return false
        if $l \leq 1$
            return true
        for all $v' \in V$
            if path_exists($v$,$v'$,$\lfloor l/2 \rfloor$) $\wedge$ path_exists($v'$,$w$,$\lceil l/2 \rceil$)
                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.

3.3.2 Polynomial space: PSPACE and NPSPACE

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

Definition 27.

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

The following inequalities should be obvious enough: $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$ (as determinism is a special case of nondeterminism), $\mathbf{P} \subseteq \mathbf{PSPACE}$ (as having polynomial time allows us to touch at most a polynomial chunk of tape), $\mathbf{NP} \subseteq \mathbf{NPSPACE}$ (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 26 (Savitch's theorem)**.** *Given a function $f(n)$,*

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

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 $N_c = 2^{O(f(|x|))}$. 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 25, 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 3.

$$\textbf{\textit{PSPACE}} = \textbf{\textit{NPSPACE}}.$$

Proof.

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

□

3.4 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 $(x_1, \dots, x_n)$, 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 28 (**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 29 (**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[3].

3.4.1 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:

[3] See also the Wikipedia articles about these classes:
`https://en.wikipedia.org/wiki/FP_(complexity)`
`https://en.wikipedia.org/wiki/FNP_(complexity)`

```
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($f|_{x_i=\top}$) = 1                        Guess the value of $x_i$
            then $x_i^* \leftarrow \top$                   Put the right truth value in the output string $x^*$
            else $x_i^* \leftarrow \bot$
        $f \leftarrow f|_{x_i=x_i^*}$                      Fix $x_i$ in $f$ to the correct truth value and simplify $f$
    return ($x_1^*, \dots, x_n^*$)
```

Similar methods can be employed also for other problems.

3.5 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[4]).

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[5].

3.5.1 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 $G'_1, \dots, G'_N$, where every G'_i is a random permutation of one of the two original graphs, chosen at random. Arthur sends the graphs $G_1, G_2, G'_1, G'_2, \dots, G'_N$ 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;
2. if G_1 and G_2 are not isomorphic, then it is possible for Merlin to tell which graphs among $G'_1, \dots, G'_N$ 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.

[4] Either Pendragon or Dent — both are fine for our purposes.

[5] For a more comprehensive exposition, see Arora-Barak, first three Sections of Chapter 8 in the online draft.

3.5.2 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:

$$\begin{aligned}
a_1 &= f(x)\\
a_2 &= g(x, a_1)\\
a_3 &= f(x, a_1, a_2)\\
a_4 &= g(x, a_1, a_2, a_3)\\
&\vdots\\
a_{2k-1} &= f(x, a_1, a_2, \dots, a_{2k-2})\\
a_{2k} &= g(x, a_1, a_2, \dots, a_{2k-1})
\end{aligned}$$

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 30. *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 $\Pr\bigl[\langle f, g\rangle_k(x)$ accepts$\bigr] \geq 2/3$;*
- *if $x \notin L$, then for any function h, $\Pr\bigl[\langle f, h\rangle_k(x)$ accepts$\bigr] \leq 1/3$.*

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.

3.5.3 The Interactive Polynomial protocol class

As an obvious generalization of the **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 30 is modified as follows:

Definition 31. *Given a language L, we say that L is decided by a* polynomial interactive proof protocol *($L \in \boldsymbol{IP}$) if there is a polynomial $k(n)$, a stochastic, polynomial function f and a (unbounded) function g such that*

- *if $x \in L$, then $\Pr\bigl[\langle f, g\rangle_{k(|x|)}(x)$ accepts$\bigr] \geq 2/3$;*
- *if $x \notin L$, then for any function h, $\Pr\bigl[\langle f, h\rangle_{k(|x|)}(x)$ accepts$\bigr] \leq 1/3$.*

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 27.

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

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 beginnning, providing the sequence $a_1, \dots, a_{2k}$ 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 $a_1, \dots, a_{2k}$ would be a polynomial certificate of acceptance. Therefore,

Theorem 28. *Let **dIP** be the "deterministic" version of **IP** where we strip stochasticity from Arthur's function f in Definition 31. Then*

$$\mathbf{dIP} = \mathbf{NP}.$$

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.

Chapter 4

Selected examples

4.1 NP-completeness proofs

4.1.1 SUBSET SUM

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

$$\sum_{i \in I} w_i = s. \tag{4.1}$$

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

Theorem 29. *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 $|x| = O(n \log \max\{x_1, \dots, x_n, s\})$. A suitable certificate is a list of indices $1 \le i_1 < i_2 < \cdots < i_k \le n$, of size $O(k \log n)$, which is linearly bounded by $|x|$. Checking the certificate requires $k \le n$ 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:

$$\begin{aligned} t_i &= a_1 a_2 \cdots a_n p_1 p_2 \cdots p_m, \\ f_i &= a_1 a_2 \cdots a_n q_1 q_2 \cdots q_m, \end{aligned} \tag{4.2}$$

where the digits of the numbers t_i and f_i are:

$$\begin{aligned} a_j &= \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise,} \end{cases} \\ p_j &= \begin{cases} 1 & \text{if the } j\text{-th clause contains } x_i \text{ without negation} \\ 0 & \text{otherwise,} \end{cases} \\ q_j &= \begin{cases} 1 & \text{if the } j\text{-th clause contains } \neg x_i \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{4.3}$$

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 $x_1, \dots, x_n$ 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

$$s = \overbrace{11\cdots1}^{n\text{ digits}} s_1 s_2 \cdots s_m, \qquad \text{with } s_1, \dots, s_m \neq 0. \tag{4.4}$$

In order to obtain a proper instance of SUBSET SUM, we need to transform (4.4) 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:

$$u_i = v_i = \overbrace{00\cdots0}^{n\text{ digits}} d_1 d_2 \cdots d_m, \tag{4.5}$$

with

$$d_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases} \tag{4.6}$$

. Therefore, F has a satisfying truth assignment if and only if we can find a subset among the numbers $t_1, \dots, t_n, f_1, \dots, f_n, u_1, \dots, u_m, v_1, \dots, v_m$ as defined in (4.2), (4.3), (4.5), (4.6), whose sum is

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

□

4.1.2 KNAPSACK

Definition 33 (KNAPSACK)**.** *Given a set of n items with weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$, 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 $I \subseteq \{1, \dots, n\}$ such that

$$\sum_{i\in I} w_i \leq c \qquad \textit{and} \qquad \sum_{i \in I} v_i \geq s. \tag{4.7}$$

The first constraint ensures that the knapsack is not going to break, the second one ensures that we have 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 $(x_1, \dots, x_n, s)$ 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 (4.7) to the equality (4.1). Therefore,

Theorem 30. *KNAPSACK is* ***NP****-complete.*

4.2 Applications of NP-completeness results

4.2.1 The Merkle-Hellman cryptosystem

Now broken, it was one of the earliest public-key cryptosystems[1] together with RSA.

Description

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

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

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 ($y_1, \dots, y_n$, $s$)     $y_1, \dots, y_n$ is super-increasing
    $I \leftarrow \emptyset$
    for i ← n..1                                  scan items starting from the largest
        if $y_i$ < $s$                            every time an item can be subtracted
            $s \leftarrow s$ - $y_i$                subtract it
            $I \leftarrow I \cup \{i\}$             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:

$$x_i \equiv y_i \cdot r \pmod{m}. \tag{4.8}$$

[1] 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 $x_1, \dots, x_n$ as her public key.

When Bob wants to send a message to Alice, he encodes it into an n-bit string $(b_1, \dots, b_n)$. 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 $\{0, \dots, m-1\}$. 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 (4.8). Since she knows r and m, which are kept secret, she can compute the inverse of r modulo m, i.e., the only number $r' \in \{1, \dots, m-1\}$ 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[2].

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

$$\texttt{SUPERINCREASING\_SUBSET\_SUM}\ (y_1, \dots, y_n, s)$$

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

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.

4.3 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

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

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 $C = (c_1, c_2, \dots, c_n) \in \{1, 2, 3\}^n$ (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 $\pi C = (\pi c_1, \dots, \pi c_n)$ 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 $s_1, \dots, s_n$ of suitable length (a few hundred bits) and will send to Alice the n values

$$d_i = H(\pi c_i \oplus s_i), \qquad i = 1, \dots, n, \tag{4.9}$$

where $\oplus$ is the concatenation operator (the bit string representing πc_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 $d_1, \dots, d_n$, 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 πc_i, πc_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 (4.9) 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 πc_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 reture 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 5

Further directions

Here are a few topics that might be interesting and that we could not discuss for lack of time.

5.1 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 $\mathbf{P} \neq \mathbf{NP}$, the average complexity of some **NP**-complete languages might be polynomial. Moreover, for some problems, hard instances might actually exist but be too hard to find.

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.

5.2 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.

5.3 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 II

Questions and exercises

Appendix A

Self-assessment questions

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

A.1 Computability

A.1.1 Recursive and recursively enumerable sets

1. Why is every finite set recursive?
   (Hint: we need to check whether n 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 $n \not\in S$? Is there a moment when we are sure that n will never be listed by $\mathcal{A}$?)

A.1.2 Turing machines

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

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

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

A.2 Computational complexity

A.2.1 Definitions

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

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

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

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?

2. What guarantees does **RP** add to make its languages more tractable than generic **NP** languages?

3. Would the functional versions of space classes make sense? Think of an example of a functional problem in **FL** (functional version of **L**).

4. Discuss the following: **FP** is precisely the class of polynomial reductions that we always used to define **NP** completeness.

5. To help your intuition about interactive proof systems, see the following Youtube video:
   `https://youtu.be/q5tEQ-WlPqo`

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.

Exercise 1

For each of the following classes of Turing machines, decide whether the halting problem is computable or not. If it is, outline a procedure to compute it; if not, prove it (usually with with a reduction from the general halting problem). Unless otherwise stated, always assume that the non-blank portion of the tape is bounded, so that the input can always be finitely encoded if needed.

1.1) TMs with 2 symbols and at most 2 states (plus the halting state), starting from an empty (all-blank) tape.

Note: in early versions the tape wasn't mentioned. My bad.

1.2) TMs with at most 100 symbols and 1000000 states.

1.3) TMs that only move right;

1.4) TMs with a circular, 1000-cell tape.

1.5) TMs whose only tape is read-only (i.e., they always overwrite a symbol with the same one);

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

Solution 1

The following are minimal answers that would guarantee a good evaluation on the test.

1.1) The definition of the machine meet the requirements for the Busy Beaver game; Since we know the BB for up to 4 states, it means that every 2-state, 2-symbol machine has been analyzed on an empty tape, and its behavior is known. Therefore the HP is computable for this class of machines.

1.2) As we have seen in the lectures, 100 symbols and 1,000,000 states are much more than those needed to build a universal Turing machine $\mathcal{U}$. If this problem were decidable by a machine, say $\mathcal{H}_{1,000,000}$, then we could solve the general halting problem "does $\mathcal{M}$ halt on input s" by asking $\mathcal{H}_{1,000,000}$ whether $\mathcal{U}$ would halt on input (M, s) or not. In other words, we could reduce the general halting problem to it, therefore it is undecidable.

1.3) If the machine cannot visit the same cell twice, the symbol it writes won't have any effect on its future behavior. Let us simulate the machine; if it halts, then we output 1. Otherwise, sooner or later the machine will leave on its left all non-blank cells of the tape: from now on, it will only see blanks, therefore its behavior will only be determined by its state. Take into account all states entered after this moment; as soon as a state is entered for the second time, we are sure that the machine will run forever, because it is bound to repeat the same sequence of states over and over, and we can interrupt the simulation and output 0; if, on the other hand, the machine halts before repeating any state, we output 1.

1.4) As it has a finite alphabet and set of states (as we know from definition), the set of possible configurations of a TM with just 1000 cells is fully identified by (i) the current state, (ii) the current position, and (iii) the symbols on the tape, for a total of $|Q| \times 1000 \times |\Sigma|^1000$ configurations. While this is an enormous number, a machine running indefinitely will eventually revisit the same configuration twice. So we just need to simulate a run of the machine: as soon as a configuration is revisited, we can stop simulating the machine and return 0. If, on the other hand, the simulation reaches the halt state, we can return 1.

1.5) Let $n = |Q|$ be the number of states of the machine. Let us number the cells with consecutive integer numbers, and consider the cells a and b that delimit the non-null portion of the tape. Let us simulate the machine. If the machine reaches cell $a-(n+1)$ or $b+n+1$, we will know that the machine must have entered some state twice while in the blank portion, therefore it will go on forever: we can stop the simulation and return 0. If, on the other hand, the machine always remains between cell $a-n$ and $b+n$, then it will either halt (then we return 1) or revisit some already visited configuration in terms of current cell and state; in such case we know that the machine won't stop because it will deterministically repeat the same steps over and over: we can then stop the simulation and return 0.

Exercise 2

For each of the following properties of TMs, say whether it is semantic or syntactic, and prove whether it is decidable or not.

2.1) M recognizes all words with an 'a' in them.
2.2) M always halts within 100 steps.
2.3) M either halts within 100 steps or never halts.
2.4) M accepts all words from the 2018 edition of the Webster's English Dictionary.
2.5) M never halts in less than 100 steps.
2.6) M is a Turing machine.
2.7) M recognizes strings that encode a Turing machine (according to some predefined encoding scheme).
2.8) M is a TM with at most 100 states.

Solution 2

2.1) 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.*

2.2) 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.

2.3) 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".

2.4) 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 2.1, the language "all words in Webster's"* is *computable, but we aren't able to always decide whether a TM recognizes it or not.*

2.5) This is the complement of property 2.2, therefore not semantic and decidable.

2.6) 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.

2.7) 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.

2.8) 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 3

3.1) Complete the proof of Theorem 8 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.
3.2) Test it for n=3.

Solution 3

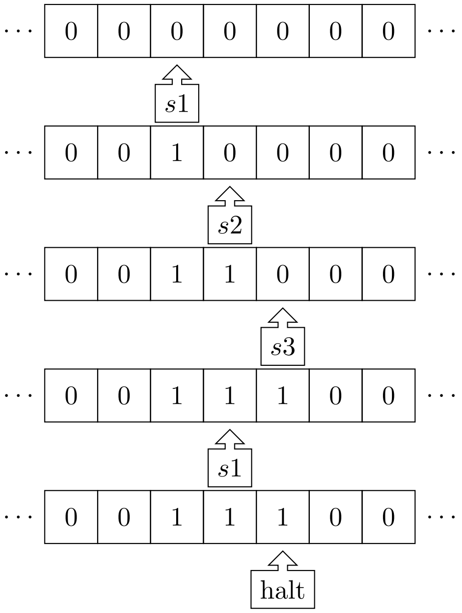
3.1) Here is a possible solution:

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

Entries marked by "—" are irrelevant, since they are never used. Any state can be used for the final move.
3.2) 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:



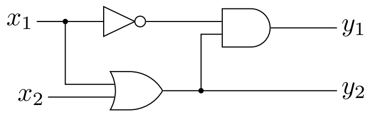
Exercise 4

Show that SAT $\leq_p$ ILP by a direct reduction.

Hint — *Given a CNF formula f, represent its variables as variables in an integer program. Use constraints to force every variable in $\{0,1\}$ and other constraints to force every clause to have at least one true literal.*

Exercise 5

Consider the following Boolean circuit:



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

Exercise 6

Consider the SET PACKING problem: given n sets $S_1, \dots, S_n$ and an integer $k \in \mathbb{N}$, are there k sets $S_{i_1}, \dots, S_{i_k}$ that are mutually disjoint?
6.1) Prove that SET PACKING $\in$ **NP**.
6.2) Prove that SET PACKING is **NP**-complete.

Hint — *You can prove the completeness by reduction of INDEPENDENT SET.*

Exercise 7

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 8

Show that **P** $\subseteq$ **ZPP**.

Hint — *A polynomial-time language is automatically in **ZPP** because...*

Exercise 9

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 10

Show that **BPP** $\subseteq$ **PP**.

Hint — *The conditions for a language to belong to a class automatically satisfy those for the other.*

Exercise 11

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 $i, j = 0, \dots, 3$ 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 12

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 13

Let SMALL SUBSET SUM[n] 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 SMALL SUBSET SUM[1000000] be still in **NP**? Would it be **NP**-complete? Would it fall back to **P**?

Exercise 14

Let SHORT SUBSET SUM[M] be a restriction of SUBSET SUM to only instances where all numbers $x_1, \dots, x_n$ in the set (apart from 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 SHORT SUBSET SUM[1000000] be still in **NP**? Would it be **NP**-complete? Would it fall back to **P**?

Exercise 15

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.
15.1) 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.
15.2) Is there a polynomial-space algorithm? Can we do any better? Describe the most space-efficient implementation that you can think of.