Foundations of Computer Science
Logic, models, and computations

Chapter: Time complexity

Course CSC_41012_EP

of l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version of August 20, 2024

Time complexity

This chapter is focusing on some particular resource of an algorithm: The times it takes to be executed.

The previous chapter applies in particular to this measure: The computation time of some algorithm is defined as the time it takes to be executed.

To illustrate the importance of this measure of complexity, let us focus on the time corresponding to algorithms of complexity n, $n \log_2 n$, n^2, n^3, 1.5^n, 2^n and $n!$ for input of size n, for increasing n, on a processor able to execute one million of elementary instructions by second. We write ∞ in the array as soon as values more than 10^{25} years (this figure is repeated from [Kleinberg & Tardos, 2006]).

Complexity	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} years
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 years	tl
$n = 100$	< 1 s	< 1 s	< 1 s	$1s$	$12,9$ years	10^{17} years	tl
$n = 1000$	< 1 s	< 1 s	1s	18 min	tl	tl	tl
$n = 10000$	< 1 s	< 1 s	2 min	12 days	tl	tl	tl
$n = 100000$	< 1 s	2 s	3 hours	32 years	tl	tl	tl
$n = 1000000$	1s	20s	12 days	$31,710$ years	tl	tl	tl

As one can see, an algorithm of exponential complexity is very rapidly useless, and is hence *not reasonable*. The main subject of this chapter is to understand what is called a *reasonable* algorithm in theoretical computer science, and to understand the theory of NP-completeness that allows to discuss the frontier between reasonable and non-reasonable algorithms.

1 The notion of reasonable time

1.1 Convention

For several reasons, the following convention has been adopted in Computer Science:

Definition 1 (Efficient algorithm) *An algorithm is* efficient *if its time complexity is polynomial, that is to say in $\mathcal{O}\left(n^k\right)$ for some integer k.*

This is a convention (and others could have been chosen[1]) that has been widely been accepted since the 70's.

Remark 1 *One can argue that an algorithm of complexity $\mathcal{O}\left(n^{1794}\right)$ is not* very reasonable. *Yes, but one must indeed fix a convention, and this is indeed considered as reasonable in theory of complexity.*

Remark 2 *Why don't taking a linear time, or a quadratic time as the notion of "reasonable": because this is not working so well. See Remark 3 below.*

1.2 First reason: To abstract from coding issues

One of the reasons for this convention is the following remark: Most of the computer science objects can be represented in various manners, but transforming one representation into the other is doable in a time that remains polynomial in the size of the encoding.

The class of polynomial being stable by composition, this implies that an algorithm that is polynomial with respect to a given representation can be transformed into a polynomial algorithm with respect to another representation.

One can then talk about *efficient* algorithm on these objects without having to go to the details on how these objects are actually represented.

Example 1 *A graph can be represented by a matrix, its* adjacency!matrix*: if the graph has n vertices, one considers an array T of size n by n, whose element $T[i][j] \in \{0,1\}$ values 1 if and only if there is an edge between the vertex i and the vertex j.*

One can also represent a graph by an adjacency!list*: To represent a graph with n vertices, one considers n lists. The list number i encodes the neighbours of vertex number i.*

One can go from one representation to the other in a time that is polynomial in the size of each: This is left to the reader to get convinced of this fact in her or his preferred programming language.

An efficient *algorithm for one of the representation can always be transformed into an* efficient *algorithm for the other representation: One just needs to start by possibly converting the representation to the representation on which the algorithm works.*

Furthermore, for the same reasons, as all of these graph representations remain polynomial in n, by using the fact that a graph with n vertices has at most n^2 vertices, an algorithm polynomial in n is nothing but an efficient algorithm on graphs, that is to say on any of the previous representations, or any usual representations of graphs.

[1] And actually, there were others previously.

1.3 Second reason: To abstract from the computational model

Another deep reason is the following: Let's come back to Chapter 7. We have proved that all computational models considered in this latter chapter can simulate one the other: RAM machines, Turing machines, 2 stacks machines, counters machines.

If we put aside the counters machines whose simulation is particularly inefficient, and whose interest is perhaps mainly only theoretical, we can observe that a number t of instructions for one model can be simulated using a number polynomial in t of instructions for the other. The class of polynomial being stable by composition, this implies that, possibly by simulating one model by the other, an algorithm that is polynomial in one model of computation can be transformed into a polynomial algorithm for any of the other models of computation.

We can then talk about *efficient* algorithms on an object without having to precise if the program is considered in one model of computation or the other[2]

In particular the notion of efficient algorithm is independent of the chosen programming language: An efficient algorithm in CAML is an efficient algorithm in JAVA, or an efficient algorithm in C.

Remark 3 *We come back to Remark 2. Why don't taking a linear time, or a quadratic time as the notion of "reasonable": In particular, because these notions of linear and quadratic time would not satisfy the above property. Indeed, the notion of linear time or of quadratic time is depending on the chosen model of computation, contrary to the notion of polynomial time computability, and/or are not closed by so nice closure properties.*

For example, for linear time, a time T for a Turing machine with two tapes is not clearly simulated in a time linear in T (This is $\mathcal{O}\left(T^2\right)$, that is quadratic with the obvious technique detailed in previous chapters, hence not linear. Notice that if this is possible to prove that $\mathcal{O}\left(T\log(T)\right)$ is possible if using a smart divide and conquer technique).

For example, for quadratic time: As $(T^2)^2 = T^4$, quadratic time is not closed by composition, hence composing a quadratic time "reasonable" algorithm with a quadratic time "reasonable" algorithm would not be "reasonable".

Since the notion of efficiency is not depending of the model, one will use the Turing machine model in all what follows: When w is a word, remember that we write length(w) for its length.

Definition 2 (TIME(t(n))) *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define the class* TIME(t(n)) *as the class of problems (languages) decided by a Turing machine in time $\mathcal{O}(t(n))$, where $n =$ length(w) is the size of the input.*

[2] Most purist will observe a problem with the RAM model of Chapter 7: One must take into account in the complexity measure the size of the integers involved in the executed elementary operations and not only the number of instructions. But this is only details, and what is written above remains totally true, is one forbids to RAM machines to manipulate integers of arbitrary size. Notice that this would anyway not be reasonable with respect to the processors that they intend to model that work in practise on words with a finite number of bits (typically 32 or 64 bits for example).

If one prefers, $L \in \mathrm{TIME(t(n))}$ if there exists a Turing machine M such that:

- M decides L: for any word w, M accepts w if and only if $w \in L$, and M rejects w if and only if $w \not\in L$;
- M takes a time bounded by $\mathcal{O}(t(n))$:
  - if one prefers: There are integers n_0, c, and k such that for every word w, if w is of sufficiently big size, that is to say if $\mathrm{length}(w) \geq n_0$, then M accepts or rejects using at most $c * t(n)$ steps, where $n = \mathrm{length}(w)$ denotes the length of w.

Remember that we focus in this chapter and in the following (and more generally in complexity theory) uniquely on decidable problems.

1.4 Class P

The class of problems which admit a reasonable algorithm corresponds then to the following class:

Definition 3 (Class P) *The class P is the class of problems (languages) defined by:*

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME(n^k)}.$$

In other words P is exactly the class of problems that admit a polynomial algorithm.

Here a several examples of problems in P.

Example 2 (Testing the colouring of a graph)

Input: *A graph $G = (V, E)$, a finite set C of colours, and colour $c(v) \in C$ for every vertex $v \in V$.*

Answer: *Decide if G is* coloured *with these colours: that is to say if there are no edge of G with two extremities of the same colour.*

This problem is in class P. Indeed, it is sufficient to go through the edges of the graph and to test for each of these edges if the colour of its two extremities are the same.

Example 3 (Evaluation in propositionnal calculus)

Input: *A formula $F(x_1, x_2, \cdots, x_n)$ of propositional calculus, some values $x_1, \cdots, x_n \in \{0,1\}$ for each of the variables of the formula.*

Answer: *Decide whether the formula F evaluates to true for this value of these vari-*

ables.

This problem is in class P. *Indeed, given some formula of propositional calculus $F(x_1,\cdots,x_n)$ and some values $x_1,x_2,\cdots,x_n \in \{0,1\}$, it is easy to compute the truth value of $F(x_1,\cdots,x_n)$. This is done in a time that one can easily check to be polynomial in the size of the input: Basically, one evaluates the formula inductively by propagating the truth value of variables and constants through logical operators (and, or, and negations) of the formula.*

Many other problems are in P.

2 Comparing problems

2.1 Motivation

It turns out however that there is a whole class of problems for which we did not succeed up to today to prove formally that this is not possible.

This is historically what leaded to consider the class of problems that we call NP, that we will consider in the following sections.

Some example of problems in this class are the following:

Example 4 (k-COLORABILITY)

Input: *A graph $G=(V,E)$ and some integer k.*

Answer: *Decide if there exists a colouring of the graph using at most k colours: that is to say decide if there exists a way to colour the vertices of G with at most k colours to obtain a colouring of G.*

Example 5 (SAT**, Satisfaction in proposititionnal calculus)**

Input: *A formula $F(x_1,\cdots,x_n)$ of propositional calculus.*

Answer: *Decide if F is satisfiable: that is to say if there exists $x_1,\cdots,x_n \in \{0,1\}^n$ such that F evaluates to true with this value of the variables $x_1,\cdots,x_n$.*

Example 6 (HAMILTONIAN CIRCUIT)

Input: *A graph $G=(V,E)$ (non-oriented).*

Answer: *Decide if there exists a* Hamiltonian circuit *in G, that is to say decide if there exists a path that goes through, once and exactly once, every vertex* *and that comes back to its starting point.*

For the three problems, some exponential time algorithm is known: test all the ways to colour the vertices for the first, or all the values of $\{0,1\}^n$ for the second, or all the paths for the last one. For the three problems, one does not know any efficient algorithm, and one has not succeeded to prove that there is none at this date.

As we will see, one can however prove that these three problems are equivalent with respect to their level of difficulty, and this will lead to consider the notion of reduction, that is to say to compare the hardness of problems.

Before, let's precise a few things.

2.2 Remarks

In this chapter and in the next chapter, we will essentially only talk about decision problems, that is to say about problems whose answer is either "true" or "false": See Definition 9.2.

Example 7 *"Sort n numbers" is not a decision problem: The output is a list of sorted numbers.*

Example 8 *"Given a graph $G=(V,E)$, determine the number of colours to colour G" is not a decision problem, as the output is some integer. One can however formulate this problem as a decision problem, of type "Given a graph $G=(V,E)$, and some integer k, determine if the graph G admits a colouring with less than k colours": This is the problem* k-COLORABILITY.

Before talking about reductions, we must talk about functions computable in polynomial time: This is the expected notion, even if we are force to provide the details as we have not done it yet.

Definition 4 (Function computable in polynomial time) *Let Σ and Σ' be two alphabets. A (total) function $f:\Sigma^* \to \Sigma'^*$ is* computable!in polynomial time *if there exists a Turing machine Turing A, working over alphabet $\Sigma \cup \Sigma'$, and some integer k, such that for every word w, A with input w terminates in time $\mathcal{O}(n^k)$ with, at the moment it stops, $f(w)$ written on its tape, where $n = \mathrm{length}(w)$.*

The following result is easy to establish:

Proposition 1 (Stability by composition) *The composition of two functions computable in polynomial time is computable in polynomial time.*

2.3 The notion of reduction

This permits to introduce the notion of reduction between problems (similar to the one of chapter 9, except that we are talking about computability in polynomial time

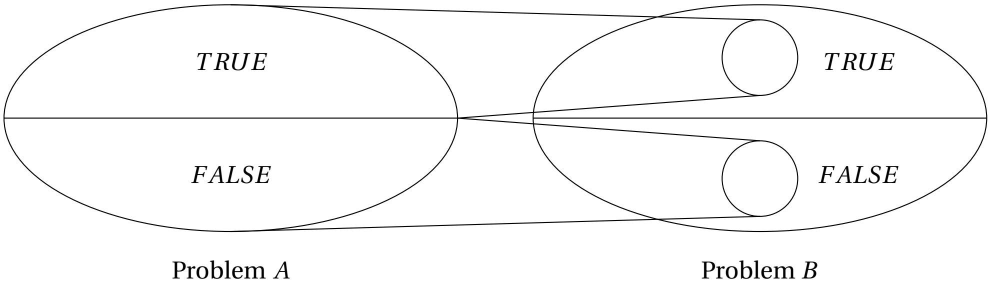


Figure 1: The reductions transform positive instances to positive instances and negative instances to negative instances.

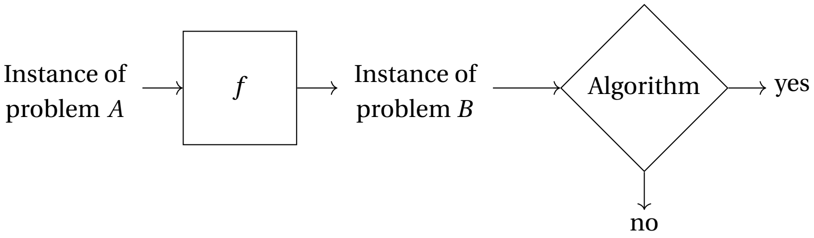


Figure 2: Reduction from problem A to problem B. If one can solve problem B in polynomial time, then one can solve problem A in polynomial time. The problem A is hence at least as easy as problem B, denoted by $A \leq B$.

instead of just computability): the idea is that if A reduces to B, then problem A is at least as easy as problem B, or if one prefers, the problem B is at least as hard as problem A: See Figure 2 and Figure 1.

Definition 5 (Reduction) *Let A and B two problems of respective alphabets Σ_A and Σ_B. A* reduction from A to B *is a function $f : \Sigma_A^* \to \Sigma_B^*$ computable in polynomial time such that $w \in A$ if and only if $f(w) \in B$. We write $A \leq B$ when A reduces to B.*

This behaves as expected: A problem is at least as easy (and hard) as itself, an the relation "being at least as easy as" is transitive. In other words:

Theorem 1 *$\leq$ is a preorder:*

1. *$L \leq L$;*

2. *$L_1 \leq L_2$, $L_2 \leq L_3$ implies $L_1 \leq L_3$.*

Proof: Consider the identity function as function f for the first point.

For the second point, suppose $L_1 \leq L_2$ via the reduction f, and $L_2 \leq L_3$ via the reduction g. We have $x \in L_1$ if and only if $g(f(x)) \in L_2$. It is then sufficient to see that $g \circ f$, provides the reduction, and is computable in polynomial time, since it is the composition of two functions computable in polynomial time. □

Remark 4 *It is not an order, since $L_1 \leq L_2$, $L_2 \leq L_1$ does not imply $L_1 = L_2$.*

It is then natural to introduce:

Definition 6 *Two problems L_1 and L_2 are equivalent, denoted by $L_1 \equiv L_2$, if $L_1 \leq L_2$ and if $L_2 \leq L_1$.*

We have then $L_1 \leq L_2$, $L_2 \leq L_1$ implies $L_1 \equiv L_2$.

2.4 Applications to comparison of hardness

Intuitively, if a problem is at least as easy as a polynomial problem, then it is polynomial. Formally:

Proposition 2 (Reduction) *If $A \leq B$, and if $B \in \mathrm{P}$ then $A \in \mathrm{P}$.*

Proof: Let f be a reduction from A to B. A is decided by the Turing machine that, on some input w, compute $f(w)$ and then simulates the Turing machine that decides B on input $f(w)$. Since we have $w \in A$ and only if $f(w) \in B$, the algorithm is correct, and it works in polynomial time. □

By considering the contrapositive of previous proposition, we obtain the following formulation that says that if a problem has no polynomial algorithm, and it is at least as easy as another one, then this latter has also no polynomial algorithm.

Proposition 3 (Reduction) *If $A \leq B$, and if $A \not\in \mathrm{P}$ then $B \not\in \mathrm{P}$.*

Example 9 *We will see that the problems* k-COLORABILITY, SAT *and* HAMILTONIAN CIRCUIT *are equivalent (and are equivalent to all the* NP*-complete problems). There is hence an efficient algorithm for one of them if and only if there is one for the other(s).*

2.5 Hardest problems

If one considers a class of problems, we can introduce the notion of hardest problem for the class, i.e. maximum for ≤. This is the notion of *completeness*:

Definition 7 ($\mathscr{C}$-completness) *Let $\mathscr{C}$ be class of decision problems.*

A problem A is said to be $\mathscr{C}$-complete, if

1. *it is in class $\mathscr{C}$;*

2. every problem B of $\mathscr{C}$ is such that $B \leq A$.

We say that a problem A if $\mathscr{C}$*-hard* it it satisfies condition 2. of Definition 7. A problem A is hence $\mathscr{C}$-completeness if it is $\mathscr{C}$-hard and in class $\mathscr{C}$.

A $\mathscr{C}$-complete problem is hence the most difficult, or one of the most difficult problems of the class $\mathscr{C}$. Clearly, if there are several, they are all equivalent:

Corollary 1 *Let $\mathscr{C}$ be a class of languages. All the $\mathscr{C}$-complete problems are equivalent.*

Proof: Let A and B two $\mathscr{C}$-complete problems. Apply condition 2 of Definition 7 to A with respect to $B \in \mathscr{C}$, and to B with respect to $A \in \mathscr{C}$. □

3 The class NP

3.1 The notion of verifier

The problems k-COLORABILITY, SAT and HAMILTONIAN CIRCUIT mentioned previously have a common point: While it is not clear that they admit a polynomial algorithm, it is very clear that they admit a polynomial *verifier*.

Definition 8 (Verifier) *A* verifier *for a problem A is an algorithm (i.e. Turing machine) V such that*

$$A = \{w | V \text{ accepts } \langle w, u\rangle \text{ for some word } u\}.$$

The verifier is polynomial if algorithm V decides its answer in a time polynomial in the length of w. One say that a language is polynomially verifiable *if it admits a polynomial verifier.*

The word u is called a *certificate* (sometimes also a *proof*, or a *witness*) for w. In other words, a verifier is using one more information, namely u, to check that w is in A.

Remark 5 *Observe that one can always restrict to certificates of length polynomial in the length of w, since in a time polynomial in the length of w the algorithm V will not read more than a polynomial number of symbols of the certificate.*

Example 10 *A certificate for the problem* k-COLORABILITY *is given by some colours for all the vertices.*

We will not always provide so many details, but here is the justification: Indeed, a graph G is in k-COLORABILITY is and only if one can find some word u that encodes the colours for all the vertices and all these colours provide a correct colouring: The algorithm V, i.e. the verifier, given $\langle G, u\rangle$, is only checking that the colouring corresponding to u is correct. This can be done in a time polynomial in the size of the graph: See discussion of Example 2.

Example 11 *A certificate for the problem* SAT *is constituted of a value* $x_1, \ldots, x_n \in \{0,1\}$ *for each of the variables of the formula* F*: The verifier needs only to check that these values satisfy the formula* F*. This can be done in a time polynomial in the size of the formula: See example 3.*

Example 12 *A certificate for the problem* HAMILTONIAN CIRCUIT *is constituted by a circuit. The verifier needs only to check that the circuit is Hamiltonian. This can be done in a time polynomial in the size of the graph.*

This leads to the following definition:

Definition 9 NP *is the class of problems (languages) that have a polynomial verifier.*

This class is important that it turns out that it contains an incredible number of problems of practical interest. It contains k-COLORABILITY, SAT and HAMILTONIAN CIRCUIT but also many other problems: See for example all the examples of the next chapter.

By construction, we have (as the empty word is a valid certificate for any problem of P):

Proposition 4 $\mathrm{P} \subseteq \mathrm{NP}$.

3.2 The question P = NP?

Clearly, we have either $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \subsetneq \mathrm{NP}$: See Figure 3.

The question to know if these two classes are equal or distinct is an impressive challenge. First because it is one of the unsolved questions among the most (maybe the) famous of Theoretical Computer Science that have been challenging research for the last 50 years: It has been selected in the list of the most important questions for Mathematics and Computer Science in 2000. The *Clay Mathematics Institute* offers 1 000 000 dollars to the person that will determine the answer to this question.

But mainly, if $\mathrm{P} = \mathrm{NP}$, then all the problems polynomially verifiable would be decidable in polynomial time. Most of the people think that the two classes are distinct since there are a very huge number of problems for which nobody have succeeded to provide a polynomial algorithm for more than 40 years.

It has also an impressive economical impact, since many systems, including today's cryptographic systems are based on the hypothesis that these two classes are distinct. If it is not the case, many considerations about these systems would collapse, and numerous cryptographic techniques would have been to be revisited.

3.3 Non-deterministic polynomial time

Let's first do a small parenthesis on terminology: The "N" in NP comes from *non deterministic* (and not from *not* as many often believe), because of the following result:

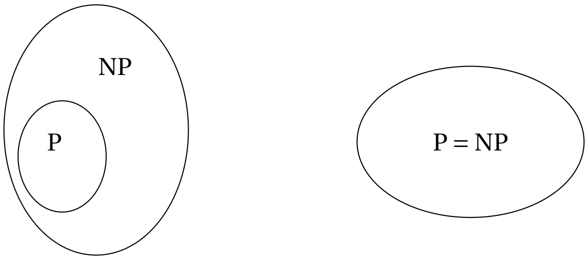


Figure 3: One of the two possibilities is correct.

Theorem 2 *A problem is in* NP *if and only if it is decided by a non-deterministic Turing machine in polynomial time.*

Remember that we have introduced the non-deterministic Turing machines in Chapter 7. We say that a language $L \subset \Sigma^*$ is *decided by the non-deterministic machine M in polynomial time* if M decides L and M takes a time bounded by $\mathcal{O}(t(n))$: There are integers n_0, c, and k such that for all words w of sufficiently big size, i.e. $n = \text{length}(w) \geq n_0$, M admits a computations that accepts in less than $c * n^k$ steps, and for $w \not\in L$, all the computations of M lead to a rejecting configuration in less than $c * n^k$ steps.

Proof: Consider a problem A of NP. Let V be the associated verifier, that runs in polynomial time $p(n)$. We build a non-deterministic Turing machine M, that, on some word w, will produce in a non-deterministic way a word u of length $p(n)$, and then will simulate V on $\langle w, u\rangle$: If V accepts, then M accepts. If V rejects, then M rejects. The machine M decides A.

Conversely, let A be a problem decided by a non-deterministic Turing machine M in polynomial time $p(n)$. As in the proof of Proposition 7.3 in Chapter 9, we can state that the non-deterministic degree of the machine is bounded by some integer r, and that the sequence of the non-deterministic choices made by the machine M up to time t can be encoded by a sequence of length t of integers between 1 and (at most) r.

Consequently, a sequence of integers of length $p(n)$ between 1 and r is a valid certificate for a word w: Given w and a word u encoding such a sequence, a verifier V can easily check in polynomial time if the machine M accepts w with this sequence of non-deterministic choices. □

More generally, we define:

Definition 10 (NTIME(t(n))**)** *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define the class* NTIME(t(n)) *as the class of problems (languages) decided by a non-deterministic Turing machine in time $\mathcal{O}(t(n))$, where n is the size of the input.*

Corollary 2

$$\mathrm{NP} = \bigcup_{k\in\mathbb{N}} \mathrm{NTIME}(\mathrm{n}^{\mathrm{k}}).$$

3.4 NP-completeness

It turns out that the class NP contains a very huge number of complete problems. The following chapter presents a whole list of such problems.

The difficulty is to succeed to produce a first such problem. This is the object of the Cook and Levin's theorem.

Theorem 3 (Cook-Levin) *The problem* SAT *is* NP*-complete.*

We will prove this theorem in the next section.

Let's first start by reformulating what this means.

Corollary 3 P = NP *if and only if* SAT ∈ P.

Proof: Since SAT is in NP, if P = NP, then SAT ∈ P.

Conversely, since SAT is NP-complete, for any problem B ∈ NP, B ≤ SAT and so B ∈ P if SAT ∈ P by Proposition 2. □

What we have just done is true for any NP-complete problem.

Theorem 4 *Let A be a* NP*-complete problem.*
P = NP *if and only if* A ∈ P.

Proof: Since A is complete it is in NP, and hence if P = NP, then A ∈ P. Conversely, since A is NP-hard, for any problem B ∈ NP, $B \leq A$ and hence B ∈ P if A ∈ P by Proposition 2. □

Remark 6 *We hence see the importance of producing* NP*-complete problems for proving* P ≠ NP*: Producing a problem for which one could succeed to prove that there is no polynomial time algorithm. At this day, none of the thousand of known* NP*-complete problems have provided a way to prove that* P ≠ NP.

Remark 7 *Remember that all the complete problems are equivalent by Corollary 1.*

3.5 A method to prove NP-completeness

The NP-completeness of a problem is established in the quasi-totality of the cases as follows:

In order to prove the NP*-completeness of a problem A, it is sufficient:*

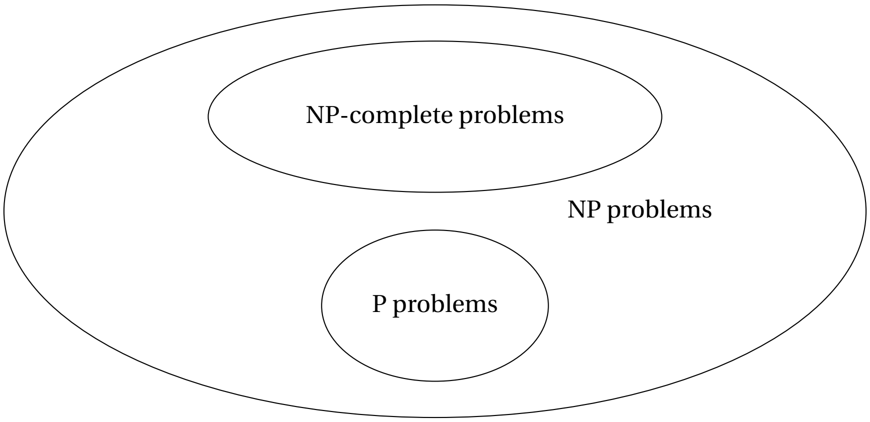


Figure 4: Situation with hypothesis P ≠ NP.

1. *to prove that it is in* NP*;*
2. *and to prove that $B \leq A$ for some problem B that is known to be* NP-*complete.*

Indeed, the point 1. guarantees that $B \in$ NP, and point 2. guarantees that for any problem $C \in$ NP we have $C \leq A$: Indeed by the NP-completeness of B we have $C \leq B$, and since $B \leq A$, we obtain $C \leq A$.

Remark 8 *Be careful, the* NP*-completeness of a problem A is obtained by proving that is is at least as hard as another* NP*-complete, and not the contrary. This is a frequent error.*

The following chapter is devoted to many applications of this strategy on various problems.

4 Two examples of proofs of NP-completeness

We apply the above strategy to prove the that 3-SAT is NP-complete.

4.1 Proof of the NP-completeness of 3-SAT

Definition 11 (3-SAT)

***Input**: A set of variables $\{x_1, \cdots, x_n\}$ and a formula $F = C_1 \wedge C_2 \cdots \wedge C_\ell$ with $C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$, where for every i, j, $y_{i,j}$ is either x_k, or $\neg x_k$ for one of the x_k.*

***Answer**: Decide whether F is satisfiable, that is, decide if there exist $x_1, \cdots, x_n \in$*

$\{0,1\}^n$ *such that F evaluates to true with this value of its variables* $x_1, \cdots, x_n$.

Theorem 5 *The problem* 3-SAT *is* NP*-complete.*

Proof: First note that 3-SAT is in NP. Indeed, given an assignment of the truth value of the variables, it is easy to check in polynomial time that the formula is true with these values of the variables.

We will reduce SAT to 3-SAT. Let F be a CNF-formula. Let C be a clause of F, say $C = x \vee y \vee z \vee u \vee v \vee w \vee t$. We introduce new variables a, b, c, d associated to this clause, and we replace C by the formula

$$(x \vee y \vee a) \wedge (\neg a \vee z \vee b) \wedge (\neg b \vee u \vee c) \wedge (\neg c \vee v \vee d) \wedge (\neg d \vee w \vee t).$$

It is easy to check that an assignment of x, y, z can be completed to an assignment of a, b, c, d that satisfies this formula if and only if C is true. By applying this construction to every clause of F, and by taking the conjunction of the formulas constructed in that way, we obtain a CNF-formula F' in which every clause has at most 3 literals whose satisfiability is equivalent to that of F.

The computation time reduces to writing the clauses, whose length is polynomial. Consequently, the whole reduction can be computed in polynomial time, and we proved, starting from SAT that 3-SAT is NP-complete. □

4.2 Proof of the NP-completeness of 3-COLORABILITY

Remember that a *colouring* of a graph is an assignment of colours to vertices of the graph such that no edge has its extremities of the same colour.

Definition 12 (3-COLORABILITY)

Input: *An undirected graph* $G = (V, E)$.

Answer: *Decide if there exists a colouring of the graph that uses at most* 3 *colours.*

Theorem 6 *The problem* 3-COLORABILITY *is* NP*-complete.*

Proof: 3-COLORABILITY is in NP, since given a colour for each of the vertices, it is easy to check in polynomial time if this is a (valid) colouring with at most 3 colours.

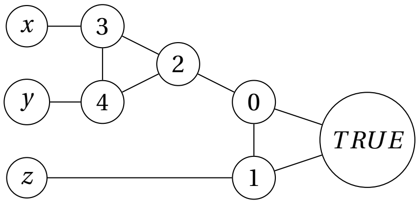
We reduce 3-SAT to 3-COLORABILITY. We hence suppose that a conjunction of m clauses with 3 literals is given, over n variables, and we have to produce a graph (with the expected properties to get a reduction). As in all reductions from 3-SAT, we have to express two constraints: first, that a variable can only take either the value 0 or 1, and second, the evaluation rules of clauses.

We construct a graph with $3 + 2n + 5m$ vertices, the first three (called distinguished vertices in what follows) are denoted by $TRUE$, $FALSE$, $DONTKNOW$.

These three vertices are linked two by two in a triangle Thus, in a colouring these three vertices must all have different colours.

We associate a vertex to every variable and to the negation of every variable. To make sure that a variable takes the value $TRUE$ or $FALSE$, for every variable x_i, we add a triangle whose vertices are x_i, $\neg x_i$, and $DONTKNOW$. This makes sure that in a colouring we must have $colour(x_i) = colour(TRUE)$ and $colour(\neg x_i) = colour(FALSE)$, or $colour(x_i) = colour(FALSE)$ and $colour(\neg x_i) = colour(TRUE)$, where of course, $colour(v)$ denotes the colour of vertex v.

It remains to encode the evaluation rules of the clauses. To do so, we introduce the following subgraph, for every clause $x \vee y \vee z$:



It can be checked that if this pattern (where the three distinguished vertices with above mentioned triangles are implicit) is 3-colourable, then the vertices 0 and 1 are $colour(FALSE)$ and $colour(DONTKNOW)$. IF 1 is $colour(FALSE)$ since a vertex corresponding to a variable must be $TRUE$ or $FALSE$, we have $colour(z) = colour(TRUE)$. If 0 is $colour(FALSE)$, then 2 cannot be $colour(FALSE)$, so 3 or 4 is, and the corresponding variable is coloured $colour(TRUE)$.

Conversely, if one of the variables is true, one can then easily construct a 3-colouration of the pattern.

Consider then the graph formed of the three distinguished vertices, of the triangles formed on these variables, and the given patterns. If this graph is 3-colourable, then in particular every subgraph is colourable. The triangles of variables are in particular colourable. From a 3-colouring of the graph, one constructs a truth assignment by setting to 1 all variables coloured with $colour(TRUE)$. This assignment is coherent (a variable and its negation have opposite values) and at least one literal for each clause is set to 1, according to the properties of the pattern above. Conversely, given an assignment of truth values, it is easy to deduce a 3-colouring of the graph.

The existence of a 3-colouring of the graph is hence equivalent to the satisfiability of the initial formula.

The reduction is clearly polynomial; hence, we have shown that 3-SAT reduces to 3-COLORABILITY. The latter is hence NP-complete. □

4.3 Proof of the Cook-Levin theorem

We cannot use the above method to prove the NP-completeness of SAT, as we do not now at this moment any NP-complete problem to reduce from.

We need to do the proof in another way, by coming back to the definition of NP-completeness: one must first prove that SAT is in NP, and second that any other problem A from NP satisfies $A \leq$ SAT.

The fact that SAT is in NP has already been established, see example 11.

Consider a problem A of NP, and an associated verifier V. The idea (which has similarities with the constructions of Chapter 10) is given a word w, to construct a formula of propositional calculus $\gamma = \gamma(u)$ which encodes the existence of an accepting computation of V on $\langle w, u\rangle$ for a certificate u.

We will build a series of formulas whose culminating point will be formula $\gamma = \gamma(u)$ that will code the existence of a sequence of configurations $C_0, C_1, \cdots, C_t$ of M such that:

- C_0 is the initial configuration of V on $\langle w, u\rangle$;
- C_{i+1} is the successor configuration of C_i, according to the transition function δ of Turing machine V, for $i < t$;
- C_t is an accepting configuration.

In other words, the existence of a valid space-time diagram corresponding to a computation of V on $\langle w, u\rangle$.

By observing that the obtained propositional formula γ remains of size polynomial in the size of w, and can indeed be obtained by a polynomial algorithm from w, we will have shown the theorem: Indeed, we will get $w \in L$ if and only if there exists u that satisfies $\gamma(u)$, that is to say $A \leq$ SAT via the function f that to w associates $\gamma(u)$.

It only remains to provide the tedious details of the construction of formula $\gamma(u)$. By hypothesis, V runs in time $p(n)$ polynomial in the size n of w. In that time, V cannot move its head more than $p(n)$ cells to the left or $p(n)$ cells to the right. We can hence restrict to a sub-rectangle of size $(2 * p(n) + 1) \times p(n)$ from the space-time diagram of the computation of V on $\langle w, u\rangle$, see Figure 5.

The cells of array $T[i, j]$ corresponding to the space-time diagram are elements of finite set $C = \Gamma \cup Q$. For every $1 \leq i \leq p(n)$ and $1 \leq j \leq 2 * p(n) + 1$ and for every $s \in C$, we define a propositional variable $x_{i,j,s}$. If $x_{i,j,s}$ has the value 1, that means that the cell $T[i, j]$ contains s.

The formula Γ is the conjunction of 4 formulas CELL $\wedge$ START $\wedge$ MOVE $\wedge$ HALT.

The formula CELL is there to guarantee that there is exactly one symbol in every cell.

$$\text{CELL} = \bigwedge_{1\leq i\leq p(n), 1\leq j\leq 2p(n)+1} \left[\left(\bigvee_{s\in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t\in C, s\neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right].$$

The symbols $\bigwedge$ and $\bigvee$ denote the iteration of corresponding symbols $\wedge$ and $\vee$. For example, $\bigvee_{s\in C} x_{i,j,s}$ is a shortcut for formula $x_{i,j,s_1} \vee \cdots \vee x_{i,j,s_l}$ if $C = \{s_1, \cdots, s_l\}$.

If we write the word $e_1 e_2 \cdots e_m$ for the word $\langle w, u\rangle$, the formula START guarantees that the first line corresponds to the initial configuration of V on $\langle w, u\rangle$.

$$\text{START} = x_{1,1,\mathbf{B}} \vee x_{1,2,\mathbf{B}} \vee \cdots \vee x_{1,p(n)+1,q_0} \vee x_{1,p(n)+2,e_1} \vee \cdots \vee x_{1,p(n)+m+1,e_m}$$

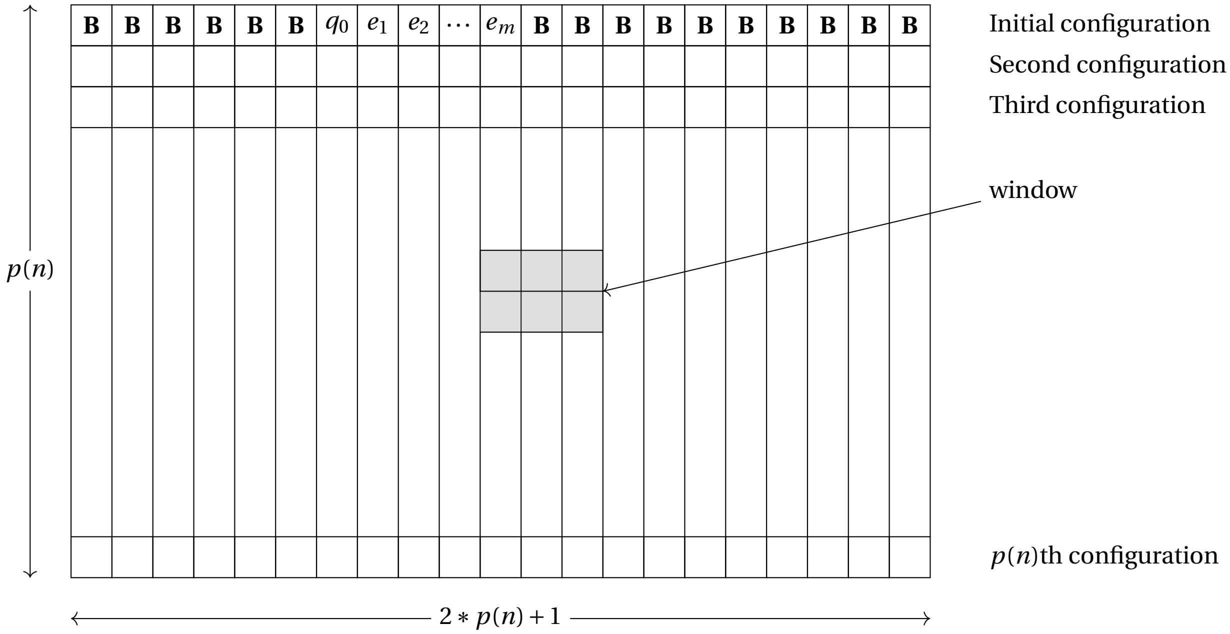


Figure 5: A $(2p(n)+1) \times p(n)$ array that codes the space-time diagram of the computation of V on $\langle w,u\rangle$.

$$\vee x_{1,p(n)+m+2,\mathbf{B}} \vee \cdots \vee x_{1,2p(n)+1,\mathbf{B}}.$$

The formula HALT guarantees that one line corresponds to an accepting configuration.

$$\text{HALT} = \bigvee_{1\le i\le p(n),1\le j\le 2p(n)+1} x_{i,j,q_a}.$$

Finally, the formula MOVE expresses that all 3 × 2 sub-rectangles from array T are legal windows: see the notion of legal window from Chapter 7.

$$\text{MOVE} = \bigwedge_{1\le i\le p(n),1\le j\le 2p(n)+1} \text{LEGAL}_{i,j},$$

where $\text{LEGAL}_{i,j}$ is a positional formula that expresses that the 3×2 subformula at position i,j is a legal window:

$$\text{LEGAL}_{i,j} = \bigwedge_{(a,b,c,d,e,f)\in\text{WINDOW}} (x_{i,j-1,a} \wedge x_{i,j,b} \wedge x_{i,j+1,c} \wedge x_{i+1,j-1,d} \wedge x_{i+1,j,e} \wedge x_{i,j+1,f}),$$

where WINDOW is the set of 6-tuple (a,b,c,d,e,f) such that if the three elements of Σ represented respectively by a,b and c appear consecutively in a configuration C_i, and if d,e,f appear consecutively at the same position C_{i+1}, then this is coherent with transition function δ of Turing machine M.

This completes the proof, noting that each of the formulas can be written easily (and hence can be produced in polynomial time from w) and that they remain of size polynomial in the size of w.

5 Some other results from complexity theory

In this section, we give several additional important results on time complexity.

5.1 Decision vs. Construction

Let us start by a remark about the hypothesis that we did on the choice of restricting to decision problems

We have talked up to now only about *decision* problems, that is problems whose answer is either true or false (for example: "given a formula F decide if the formula F is satisfiable") in contrast to problems that consist in *producing an object with a property* (for example: given a formula F, produce an assignment of the variables that makes it true if there exists one).

Clearly, producing a solution is at least as hard as deciding if there exists one, and hence if P ≠ NP, none of the two problems has a solution in polynomial time, and the same holds for any NP-complete problem.

However, if P = NP, it turns out that we can also produce a solution:

Theorem 7 *Assume that* P = NP. *Let L be a problem of* NP *and V the associated verifier. One can construct a Turing machine that on any input $w \in L$ produces in polynomial time a certificate u for w for verifier V.*

Proof: Let us start by proving the theorem for L being the satisfaction problem of propositional formula (so the problem SAT). Assume P = NP. Then one can then test if a propositional formula F with n variables is satisfiable or not in polynomial time. If it is satisfiable, then one can fix its first variable to 0 and test if the obtained formula F_0 is satisfiable. If it is, then we write 0 and then restart recursively with this formula F_0 with $n-1$ variables. Otherwise, necessarily any certificate must have its first variable set to 1. Write 1 and start recursively with formula F_1 whose first variable is fixed to 1, and that has $n-1$ variables. Since it is easy to check if a formula without any variable is satisfiable, by this method, a correct certificate will be produced.

Now if L is an arbitrary language of NP, we can use the fact that the reduction produced by the proof of the Cook-Levin theorem is a *Levin* reduction: Not only do we have $w \in L$ if and only if $f(w)$ is a satisfiable formula, but one can find a certificate for w from a certificate of the satisfiability of formula $f(w)$. One can then use the previous algorithm to find the certificate for L. □

What we used in the above proof is the fact that the satisfiability problem of a CNF-formula is *self-reducible* to instances of lower size.

5.2 Hierarchy theorems

We say that a function $f(n) \geq n\log(n)$ is *time constructible*, if the function that sends 1^n to the binary representation of $1^{f(n)}$ is computable in time $\mathcal{O}\left(f(n)\right)$.

Most of the usual functions are time constructible, and in practice this is not really a restriction.

Remark 9 *For example, $n\sqrt{n}$ is time constructible: On input 1^n, one starts by counting the number of 1 in binary. One can use for that a counter, which remains of size $\log(n)$, that one increments: This is hence done in time $\mathcal{O}\left(n\log(n)\right)$ since one uses at most $\mathcal{O}\left(\log(n)\right)$ steps for every letter of the input word. One can then compute $\lfloor n\sqrt{n}\rfloor$ in binary from the representation of n. Any standard method for doing so runs in time $\mathcal{O}\left(n\log(n)\right)$, since the size of the involved numbers is $\mathcal{O}\left(\log(n)\right)$.*

Theorem 8 (Time Hierarchy theorem) *For every time constructible function $f: \mathbb{N} \to \mathbb{N}$, there exists a language L that is decidable in time $\mathcal{O}\left(f(n)\right)$ but not in time $o(f(n)/\log f(n))$.*

Proof: The proof is a generalization of the idea of the proof of Theorem 14.11 of next chapter: We invite our reader to wait and start by this latter proof.

We prove a version weaker than the statement above. Let $f: \mathbb{N} \to \mathbb{N}$ be a time constructible function.

One considers the (very artificial) language L that is decided by the following Turing machine B:

- on an input w of size n, B computes $f(n)$ and memorize $\langle f(n)\rangle$ the binary encoding of $f(n)$ in a binary counter c;
- If w is not of the form $\langle A\rangle 10^*$, for some Turing machine A, then Turing machine B rejects.
- Otherwise, B simulates A on the word w for $f(n)$ steps to determine whether A accepts in a time less than $f(n)$:
  - If A accepts in this time, then B rejects;
  - otherwise B accepts.

In other words, B simulates A on w, step by step, and decrements the counter c at each step. If this counter reaches 0 or if A rejects, then B accepts. Otherwise, B rejects.

By the existence of a universal Turing machine, there exist integers k and d such that L is decided in time $d \times f(n)^k$.

Suppose that L is decided by a Turing machine A in time $g(n)$ with $g(n)^k = o(f(n))$. There must exists an integer n_0 such that for $n \geq n_0$, we have $d \times g(n)^k < f(n)$.

As a consequence, the simulation of A by B will indeed be complete on some input of size n_0 or more.

Consider what happens when B is run on the input $\langle A \rangle 10^{n_0}$. Since this input is of size greater than n_0, B answers the opposite of Turing machine A on the same input. Hence B and A are not deciding the same language, and hence Turing machine A is not deciding L, which leads to a contradiction.

As a consequence L is not decidable in time $g(n)$ for any function $g(n)$ with $g(n)^k = o(f(n))$.

The theorem is a generalization of this idea. The (inverse) factor $\log(f(n))$ comes from the construction of a universal Turing machine that is more efficient than the one considered in this document, introducing only a logarithmic time slow-down. □

Formulating the above theorem differently, we get:

Theorem 9 (Time Hiearchy theorem) *Let $f, f' : \mathbb{N} \rightarrow \mathbb{N}$ be time constructible functions such that $f(n)\log(f(n)) = o(f'(n))$. Then the inclusion* $\mathrm{TIME(f(n))} \subsetneq \mathrm{TIME(f'(n))}$ *is strict.*

We obtain for example:

Corollary 4 $\mathrm{TIME(n^2)} \subsetneq \mathrm{TIME(n^{\log n})} \subsetneq \mathrm{TIME(2^n)}$.

We define:

Definition 13 *Let*

$$\mathrm{EXPTIME} = \bigcup_{c \in \mathbb{N}} \mathrm{TIME}(2^{n^c}).$$

We obtain:

Corollary 5 $\mathrm{P} \subsetneq \mathrm{EXPTIME}$.

Proof: Any polynomial becomes eventually negligible smaller than 2^n, and hence P is a subset of $\mathrm{TIME(2^n)}$. Now, $\mathrm{TIME(2^n)}$, that contains all P is strict subset of, for example, $\mathrm{TIME(2^{n^3})}$, that is included in EXPTIME. □

5.3 EXPTIME **and** NEXPTIME

Consider

$$\mathrm{EXPTIME} = \bigcup_{c \geq 1} \mathrm{TIME}(2^{n^c})$$

and

$$\mathrm{NEXPTIME} = \bigcup_{c \geq 1} \mathrm{NTIME}(2^{n^c}).$$

We can prove the following result (and this is not hard):

Theorem 10 *If* EXPTIME ≠ NEXPTIME *then* P ≠ NP.

6 One the meaning of the $P = NP$ question

We make a digression about the meaning of the P vs. NP question in relation to proof theory and other chapters of this document.

One can see NP as the class of languages such that testing the containment to it is equivalent to determining if there is a *short* (polynomial size) certificate. This can be related to the existence of a proof in mathematics. Indeed, in its own principle, mathematical deduction consists in proving theorems starting from axioms.

One expects that the validity of a proof is easy to check: one only needs to check that each line of the proof is a consequence of the previous lines, in the proof system. Actually, in most of the axiomatic proof systems (for example in all the proof systems we have seen) this verification can be done in a time that remains polynomial in the length of the proof.

Consequently, the following decision problem is NP for all the particular axiomatic usual proof systems $\mathscr{A}$, and in particular for the one $\mathscr{A}$ that we have seen for the predicate calculus.

$$\text{THEOREMS} = \{\langle\phi, \mathbf{1}^n\rangle | \phi \text{ has a proof of length } \leq n$$
$$\text{in system } \mathscr{A}\}.$$

We leave to our reader the following exercise:

Exercise 1 *The set theory of* Zermelo-Fraenkel *is one of the axiomatic systems that allows axiomatizing mathematics with a finite description. (Even without knowing all the details of the set theory of* Zermelo-Fraenkel*) argue at a high level that the problem* THEOREMS *is* NP*-complete for the set theory of* Zermelo-Fraenkel.

Hint: the satisfiability of a Boolean circuit is particular statement.

In other words, in virtue of Theorem 7, the P = NP question is the one (that has been asked for the first time by Kurt Gödel) to know whether there exists a Turing machine that is able to produce a mathematical proof of all statements ϕ in a time polynomial in the length of its proof.

Does this seem reasonable?

What is the meaning of the NP = coNP **question?** Remember that coNP is the class of languages whose complement is in NP. The question NP = coNP, is related to the existence of short proofs (of certificates) for statements that do not seem to have one: for example, it is easy to prove that propositional formula is satisfiable (one produces a valuation of its inputs, that one can encode in a proof that says that by propagation of the inputs towards the outputs, that the circuit outputs 1). On the other hand, in the general case, it is not clear how to write a short proof that a given propositional formula is not satisfiable. If NP = coNP, there must always exist one: The question is related to the existence of way proving the non-satisfiability of a propositional formula different from usual methods.

One can formulate equivalent statements for all the mentioned NP-complete problems.

7 Exercises

Exercise 2 *Prove that class* P *is closed under union, concatenation and complement.*

Exercise 3 *Prove that class* NP *is closed under union and concatenation.*

Exercise 4 *(solution on page 234) Prove that if* NP *is different from its complement then* P $\neq$ NP.

Exercise 5 *Prove that if* P = NP *then all languages* $A \in P$ *except for* $A = \emptyset$ *and* $A = \Sigma^*$ *are* NP*-complete.*

8 Bibliographic notes

Suggested readings To go further with the notions of this chapter, we suggest to read the books [Sipser, 1997], [Papadimitriou, 1994] [Lassaigne & de Rougemont, 2004].

A reference book that contains the last results of the domain is [Arora & Barak, 2009].

Bibliography This chapter contains some standard results in complexity. We essentially used their presentation in [Sipser, 1997], [Poizat, 1995], [Papadimitriou, 1994]. The last part “discussion” is taken from [Arora & Barak, 2009].

Index

≡, 10
≤, 9, 10
coNP, 23
3-COLORABILITY, 16

adjacency
  list, 4
  matrix, 4
algorithm
  efficient, *see* efficient algorithm

$\mathscr{C}$-completeness, *see* completeness
C-hardness, *see* completeness
certificate, 11, 20, 23
CIRCUIT HAMILTONIAN, 12
COLORABILITY, 7
colouring of a graph, 6, 7, 16
completeness, 10, 11
computable
  in polynomial time, 8
computation
  time, 3
constructible
  in time , *see* time constructible function
Cook-Levin theorem, 14

efficient, 3–5, 8
  algorithm, 3
equivalence
  between problems, 11, 14
EXPTIME, 22, 23

function
  time constructible, *see* time constructible function

graph, 4
  representation, *see* adjacency list or adjacency matrix

HAMILTONIAN CIRCUIT, 11
Hamiltonian circuit of a graph, 7
hard, *see* completeness
hierarchy
  time , *see* time hierarchy theorem

k-COLORABILITY, 11, 12

language
  decided by a Turing machine
    non-deterministic, 13
legal window, 19
Levin reduction, 20

NEXPTIME, 22, 23
NP, 12–14, 23
NP, 12, 14, 17
NP-completeness, 3, 14, 16, 17
NTIME(), 13

P, 6, 12
polynomially verifiable, 11
problem, 20
  of decision, 8, 20
proof, 11

reasonable, 3, 4
reduction, 8, 9
  Levin, *see* Levin reduction

SAT, 7, 11, 12, 14, 17, 20
satisfaction
  of a formula, 15, 16
satisfiability
  of a formula, 7

satisfiable
    (for a formula), *see* satisfiability of a formula
self-reducible problem, 20

TIME(), 5
time
    constructible function, 21
    hierarchy theorem, 21, 22
Turing machine
    non-deterministic, 13

verifier, 11

witness, 11

Zermelo-Fraenkel, 23

Bibliography

[Arora & Barak, 2009] Arora, S. & Barak, B. (2009). *Computational Complexity: A Modern Approach.* Cambridge University Press. `https://doi.org/10.1017/cbo9780511804090`

[Kleinberg & Tardos, 2006] Kleinberg, J. M. & Tardos, É. (2006). *Algorithm design.* Addison-Wesley.

[Lassaigne & de Rougemont, 2004] Lassaigne, R. & de Rougemont, M. (2004). *Logic and complexity.* Discrete Mathematics and Theoretical Computer Science. Springer. `https://doi.org/10.1007/978-0-85729-392-3`

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

[Poizat, 1995] Poizat, B. (1995). *Les petits cailloux: Une approche modèle-théorique de l'Algorithmie.* Aléas Editeur.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation.* PWS Publishing Company.