2

THE NOTIONS OF FEASIBILITY AND NP-HARDNESS: BRIEF INTRODUCTION

The main goal of this book is to analyze computational complexity and feasibility of data processing and interval computations. In Chapter 1, we defined the basic problems of data processing and interval computations. In this chapter, we give a brief introduction to the notions related to feasibility and computational complexity.

2.1. When is an Algorithm Feasible?

2.1.1. *What Does "Feasible" Mean? The Main Idea*

Some algorithms are not feasible. In theory of computation, it is well known that not all algorithms are feasible (see, e.g., Garey *et al.* [120], Lewis *et al.* [250], Martin [273]): whether an algorithm is feasible or not depends on how many computational steps it needs.

For example, if for some input x of length $\text{len}(x) = n$, an algorithm requires 2^n computational steps, then for an input of a reasonable length $n \approx 300$, we would need 2^{300} computational steps. Even if we use a hypothetical computer for which each step takes the smallest physically possible time (the time during which light passes through the smallest known elementary particle), we would still need more computational steps than can be performed during the (approximately 20 billion years) lifetime of our Universe.

A similar estimate can be obtained for an arbitrary algorithm whose running time $t(n)$ on inputs of length n grows at least as an exponential function, i.e.,

for which, for some $c > 0$, $t(n) \geq \exp(c \cdot n)$ for all n. As a result, such algorithms (called *exponential-time*) are usually considered *not feasible*.

Comment. The fact that an algorithm is not feasible, does not mean that it can never be applied: it simply means that there are cases when its running time will be too large for this algorithm to be practical; for other inputs, this algorithm can be quite useful.

Some algorithms are feasible. On the other hand, if the running time grows only as a polynomial of n (i.e., if an algorithm is *polynomial-time*), then the algorithm is usually quite feasible.

Existing definition of feasibility: the main idea. As a result of the above two examples, we arrive at the following idea: An algorithm $\mathcal{U}$ is called *feasible* if and only if it is *polynomial-time*, i.e., if and only if there exists a polynomial $P(n)$ such that for every input x of length $\text{len}(x)$, the computational time $t_{\mathcal{U}}(x)$ of the algorithm $\mathcal{U}$ on the input x is bounded by $P(\text{len}(x))$: $t_{\mathcal{U}}(x) \leq P(\text{len}(x))$.

In most cases, this idea works. In most practical cases, this idea *adequately* describes our intuitive notion of feasibility: *polynomial-time* algorithms are usually *feasible*, and *non-polynomial-time* algorithms are usually *not feasible*.

This idea is not perfect, but it is the best we can do. Although in *most* cases, the above idea adequately describes the intuitive notion of feasibility, the reader should be warned that this idea is *not perfect*: in some (very rare) cases, it does not work (see, e.g., Garey *et al.* [120], Lewis *et al.* [250], Martin [273]):

- Some algorithms are polynomial-time but not feasible: e.g., if the running time of an algorithm is $10^{300} \cdot n$, this algorithm is polynomial-time, but, clearly, not feasible. (Other examples of such algorithms, examples that are directly related to data processing and interval computations, will be given in Chapter 4.)
- Vice versa, there exist algorithms whose computation time grows, say, as $\exp(0.000\ldots01 \cdot \text{len}(x))$. Legally speaking, such algorithms are exponential time and thus, not feasible, but for all practical purposes, they are quite feasible.

It is therefore desirable to look for a *better* formalization of feasibility, but as of now, "polynomial-time" is the best known description of feasibility.

In view of this, in the following text, we will use the terms "feasible" and "polynomial-time" interchangingly, and we will specifically mention the rare cases when these two notions differ.

2.1.2. *How Can We Formalize the Main Idea?*

How can we formalize the main idea? The above idea of polynomial-time algorithms is based on the following two fundamental notions: the *length* of the input and the *computation time*. So, to formalize the notion of feasible algorithms, we must formalize the notions of *input length* and *computation time*.

First try: computer-specific measures of input length and computation time. At first glance, these seems to be no problem with formalizing the notions of input length and computation time: if we fix a computer and if we fix a programming language, then we can define the *length* of the input, e.g., as the number of symbols on it, and the *computation time* as the actual (physical) time from the start of the program to the moment when it produces the results.

To make computer-independent conclusions about feasibility of algorithms, we need computer-independent measures of input length and computation time. The above-described "computer-specific" approach to defining input length and computation time is not perfect:

- Different languages use different symbols to describe the same data; as a result, the *length* of the input may drastically change if we change the language.
- Similarly, different computers require different *times* to perform different operations. Even on one and the same computer, the computation time for one and the same operation may change from time to time, depending, e.g., on whether the registers are currently available, whether any periodic automatic maintenance operations are currently being performed, etc.

As a result, the dependence of the computation time on the input length may drastically change from one computer to another. Hence, if we use the computer-specific measures of input length and computation time, we *will* be able to show that some algorithms *are* polynomial-time, but it will be very *difficult* to prove that some problems are *not* polynomial-time: Indeed, we can only show that the algorithm is not polynomial-time on a *given* computer, but

it is very difficult to argue that it will *not* become polynomial-time on some other computer.

If we want *computer-independent* results, we must, therefore, be able to describe *computer-independent* measures of input length and time complexity.

2.1.3. *First Step Towards Formalization of Feasibility: Computer-Independent Notion of the Input Length (Number of Bits)*

General idea. It is easy to come up with a computer-independent notion of the input length. Indeed, although computers use different hardware, most of them use the same way of representing all their data: each element of data is represented by a sequence of 0's and 1's. Thus, we can always measure the length of each input by the number of *binary units* (also called *bits*), i.e., by the number of 0's or 1's, that are needed to represent it.

Integers. In particular, inside the computer, *integers* are usually represented in their binary form. Therefore, the input length of an integer can be naturally defined as the number of bits in its binary expansion: e.g., the number $8_{10} = 1000_2$ requires 4 bits, while $26_{10} = 11010_2$ requires 5 bits.

Binary-rational numbers. Similarly, *binary-rational* numbers, i.e., numbers of the type $p/2^q$, are usually represented in their fixed-point binary form: e.g., $3/8_{10} = 0.011_2$ takes 4 bits, while $19/16_{10} = 1.0011_2$ requires 5 bits. So, e.g., if we say that "there is a polynomial-time algorithm that, for every binary-rational number ε, computes ...", we mean, in particular, that for the values $\varepsilon_n = 0.0\ldots01_2 = 2^{-n}$ of length n, the running time of this algorithm is bounded by a polynomial of n.

General rational numbers. A general *rational* number m/n, where m and n are integers, can be naturally represented as a *pair* of integers m and n; therefore, we take the total length of the binary representations of m and n as the input length: e.g., $5/7_{10} = 101/111_2$ requires 6 bits to describe.

Fixed-point and floating point numbers: a comment. In most computers, there is no special *rational* data type, there is a type *real* which actually describes binary rational numbers. In most computers, there are two *different* representations of these "real" numbers: in addition to the above-described *fixed-point* real numbers, there are also *floating-point* real numbers, in which a

binary rational number is represented as $m \cdot 2^e$, where m is a fixed-point real number (usually, with only zero before the binary point) and e is an integer.

In this book, we describe the input length in terms of the *fixed point* representation. Most of our results about computational complexity and feasibility of data processing and interval computations, both positive (that some problems can be solved by feasible algorithms) and negative (that for some other problems no feasible algorithm is possible) are true for floating point numbers as well: e.g., since every fixed point number is at the same time a floating point number (with $e = 0$), *negative* results about fixed point inputs are automatically transformed into *negative* results about the floating point inputs.

However, to avoid potential confusion, we decided, in this book, not to mention the floating point representation versions of our results at all. The reason why such a confusion can occur is that with floating point representation, there are some negative results that are caused not by a complexity of the problem, but by a *peculiarity* of this representation. Let us give a simple example of this peculiarity:

- For *fixed-point* binary-rational numbers x and y, *exactly* computing their *sum* $x + y$ is easy: the standard bit-after-bit addition requires linear time.
- However, for *floating point* binary-rational numbers, this same addition problem requires *exponential* time: e.g., if we take $x = 1$ and $y = 1 \cdot 2^e$ with $e = -2^n = -10\ldots0_2$ (n zeros), with the total length $\approx n$, then the exact description of $x + y = 1.0\ldots01$ (2^n zeros) requires *exponentially many* bits, so generating these bits would require *exponentially long* time.

This example does *not* mean that floating point representation is in any way inferior and bad: it is known to be very useful, and the above difficulty with addition can be easily overcome if we require that the result be known with a given *accuracy*. However, this example clearly shows that floating-point formulations require extra accuracy and complexity that appears even for such simple operations as addition. In other words, floating-point formulations contain extra complexity that is unrelated to the complexity of data processing and interval computations; thus, if we try to reformulate our results in terms of floating point numbers, the resulting combination of two complexities would make the corresponding results very un-intuitive.

2.1.4. *Second Step Towards Formalization of Feasibility: Computer-Independent Notion of Computation Time*

General idea. On each computer, every algorithm, every program consists of a sequence of *elementary steps*, i.e., hardware-supported simple operations such as operations with bits, or addition, etc. We can estimate the computation time by simply counting the number of these elementary steps, and multiplying this number by the average time of each step. (We may also want to take into consideration that different elementary steps take different times.) Then, if we have a similar computer, with similar (but faster) elementary steps, we can estimate the running time on a new computer if we already know the number of steps needed to run the algorithm.

This is the general idea behind different computer-independent estimates.

The situation is not so easy as it may seem. The actual estimates are somewhat more difficult to get because different computers may have different elementary steps. Let us give two examples:

- First computers only had hardware-supported *fixed-point* operations, so a floating-point operation had to be implemented as several fixed-point ones, and thus, was counted as *several* steps. In most modern computers, *floating-point* operations are also directly hardware supported and thus, each such operation can be counted as *one* step.
- In many modern computers, computing the element-wise sum $c_i \leftarrow a_i + b_i$ of two vectors $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ means n additions, i.e., n elementary steps. On the other hand, in computers with a math co-processor, the entire addition is directly hardware supported and therefore, can be counted as a single elementary step.

To get a computer-independent definition, we have to consider different types of computers and corresponding definitions.

Turing machines, and a standard complexity measure. Most textbooks on complexity theory and theory of computation start with the simplest possible "computer", that was designed by Turing long before the actual computers. This toy computer (called *Turing machine*) consists of a potentially infinite *tape* divided into sequentially located *cells*, and a *head* that moves along this

tape, from one cell to another (the reader can see that Turing machine probably resembles a tape recorder more than it resembles a modern computer).

To specify a Turing machine, we must describe three things:

- The *first* thing we must describe is the (finite) set of all possible symbols that it can place in each cell. This set (called *alphabet*) can simply consist of two possible symbols 0 and 1, in which case, each cell can be in one of the three states: nothing is written in this cell, 0 is written, and 1 is written. We can also consider Turing machines with a larger alphabet: e.g., cells can simulate *bytes* (i.e., 8-bit sequences), in which case, the alphabet consists of $2^8 = 256$ possible symbols: 0000 0000, 0000 0001, ..., 1111 1111.
- *Second*, we must describe the possible state of the head. In more precise terms, we must describe another finite set, whose elements will be called *states* (or *states of the head*). There must be two special states: *starting* state (in which we start computations) and the *halting* state, on reaching which the computer stops.
- Finally, we must describe the *action* of the computer. Namely, for every state s of the head and for every symbol σ in the cell to which the head is currently pointing, we must describe what this computer will do. It can do one of the three things:
  - It can *overwrite* the symbol σ with some other symbol σ' that, in general, depends on s and σ ($\sigma' = \sigma'(s, \sigma)$), and at the same time *change* its state into the some other state $s' = s'(s, \sigma)$;
  - It can, instead, move one step to the *right*.
  - It can also move one step to the *left*.

We start in the *starting* state, with the input written on this tape, and then *apply* the computer's *action* again and again until we reach the *halting* state. When this happens, the tape should contain the result of the computations.

Let us give a very simple *example* of a Turing machine, that adds 1 to a binary number. Since we are dealing with binary numbers, we only need the alphabet $\{0, 1\}$. The machine starts at the end of the input binary number, and proceeds as follows:

- If it is in the starting state s_0, and it sees 0, it changes it to 1 and gets into the halting state h (and stops).
- If it is in the starting state s_0, and it sees 1, it changes it to 0 and gets into the state s_1 ("ready to move, carrying a carry").
- If it is in the state s_1, then, no matter what it sees, it moves one step to the left and gets into the state s_2 ("I have a carry").
- If it is in the state s_2, and it sees 0, then it changes 0 to 1 and halts.
- If it is in the state s_2 and it sees 1, it changes 1 to 0 and goes into the state s_1.
- Finally, if it is in the state s_2 and sees an empty cell, it changes the contents of this cell to 1 and halts.

For Turing machines, we have clearly defined steps, so we can define computational complexity $t_{\mathcal{U}}(x)$ of an algorithm $\mathcal{U}$ on an input x as the number of the corresponding steps. This is the *standard definition* of computational complexity in theory of computing.

Turing machine is a very primitive computer, on which a simple operation that is hardware supported on a normal computer can take a very long time. So, the Turing machine-based complexity is a rather poor estimate for the *actual* computation time of an algorithm on a real computer. If we use more sophisticated computer models, we can get much *better* estimates.

What makes Turing machine-based complexity *standard* and widely used is the fact that although the *actual* computation time changes from one type of computer to another, but whether the algorithm is *polynomial-time* or not does not depend on our choice of the computer. Thus, whatever reasonable class of computers we consider, a problem can be solved in polynomial-time on computers of this class if and only if we can solve it in polynomial time on a Turing machine (for details, see, e.g., Emde Boas [99]). Thus, if all we are interested in is whether an algorithm is feasible or not, Turing machines are quite sufficient.

Since we are also interested in more realistic estimates of computation time, we will use more realistic computer models.

RAM and bit complexity. In Turing machines, if we are currently facing cell 1, and we want to use the contents of cell n, we actually have to move step-by-step, and spend n computation steps just to reach this new cell. This is definitely not realistic. In real computers, if we know the number of the cell, we immediately go there. In other words, we can go to an arbitrarily ("randomly") chosen cell in a single step. If we add this ability to the Turing machine, we get the so-called *RAM* (Random Access Memory) computers. For the particular case when cells can only contain 0 and 1, the number of computational steps on RAM is sometimes called *bit complexity*.

Algebraic complexity. RAM is slightly more realistic than a Turing machine, but it is still not very realistic. In real-life computers, in addition to operations with bits, we have a hardware support for elementary *arithmetic* operations such as addition and multiplication of two integers. It is therefore reasonable, given an input of length n, to assume that we can perform addition and multiplication of integers of length $C \cdot n$ (for some reasonable C) in a single step. The number of computational steps on such machine is called *algebraic complexity*.

Algebraic complexity is the closest to the actual computation time and therefore, we will use this complexity measure in the book. To be more precise:

- When we claim that something is *polynomial-time*, this claim (as we have already mentioned) will be independent on the choice of complexity measure. But:
- If we claim something more specific, like *linear time* (i.e., $t_U(n) \leq C \cdot n$) or *quadratic time* (i.e., $t_U(n) \leq C \cdot n^2$), we will mean exactly linear time (correspondingly, quadratic time) in the sense of algebraic complexity.

Comment. The relation between algebraic and bit complexity is analyzed, e.g., in Pan [321].

A remark about BSS complexity. To go from bit complexity to algebraic complexity, we counted each arithmetic operation with numbers of *fixed* length as a single computation step. We can go further and count each operation with binary-rational numbers of *arbitrary* length (or even with arbitrary *real* numbers, not necessarily rational) as a single computation step. This definition was, in effect, proposed by Blum, Shub, and Smale [49] and is called BSS complexity (see also Smale [397], Cucker *et al.* [81, 80], Meer [280], Grädel *et al.* [132], Lickteig *et al.* [254]).

For our problems, BSS complexity is very useful in proving *negative* results: If a problem is *difficult* according to this BSS complexity, then it will be even more difficult if we only allow a narrower class of operations, i.e., it will be difficult according to algebraic complexity as well.

On the other hand, if a problem is *easy* in the sense of BSS complexity, it often means that this problem is actually easy, but sometimes, it can become difficult if we only allow operations with bounded numbers; examples of such *difference* between BSS and algebraic complexity are given, e.g., in Meer [279, 281]. Because of this difference, in this book, we will not use BSS complexity.

2.1.5. *Formal Definition of Feasibility*

Now, we are ready for the precise definition:

Definition 2.1. *An algorithm U is called feasible if there exists a polynomial $P(n)$ such that for every input x, the running time $t_U(x)$ of this algorithm does not exceed $P(\text{len}(x))$, where by $\text{len}(x)$, we denoted the length of the input x (i.e., the number of bits that form this input).*

2.2. When is a Problem Tractable?

2.2.1. *Ideal Solution is not yet Possible*

What would be an ideal solution. At first glance, now, that we have a definition of a feasible algorithm, we can describe which problems are tractable and which problems are intractable: If there exists a polynomial-time algorithm that solves all instances of a problem, this problem is tractable, otherwise, it is intractable.

Sometimes, this ideal solution is possible. In some cases, this ideal solution is possible, and we either have an explicit polynomial-time algorithm, or we have a proof that no polynomial-time algorithm is possible.

Alas, for many problems, we do not know. Unfortunately, in many cases, we do not know whether a polynomial-time algorithm exists or not. This does not mean, however, that the situation is hopeless: instead of the missing *ideal*

information about intractability, we have another information that is almost as good:

What we have instead of the ideal solution. Namely, for some cases, we do not know whether the problem can be solved in polynomial time or not, but we do know that this problem is as hard as practical problems can get: if we can solve *this* problem easily, then we would have an algorithm that solves *all* problems easily, and the existence of such universal solves-everything-fast algorithm is very doubtful. We can, therefore, call such "hard" problems *intractable*.

In order to formulate this notion in precise terms, we must describe what we mean by a problem, and what we mean by the ability to *reduce* other problems to this one.

2.2.2. *How Can We Define a General Practical Problem?*

What is a practical problem: informal idea. What is a practical problem? When we say that there is a practical problem, we usually mean that:

- we have some information (we will denote its computer representation by x), and
- we know the relationship $R(x, y)$ between the known information x and the desired object y.

In the computer, everything is represented by a binary sequence (i.e., sequence of 0's and 1's), so we will assume that x and y are binary sequences.

Two examples of problems. In this section, we will trace all the ideas on two examples, one taken from mathematics and one taken from physics. Readers who do not feel comfortable with one of the example (say, with a physical one) are free to simply skip it.

- (Example from *mathematics*) We are given a mathematical statement x. The desired object y is either a proof of x, or a "disproof" of x (i.e., a proof of "not x"). Here, $R(x, y)$ means that y is a proof either of x, or of "not x".

- (Example from *physics*) x is the results of the experiments, and the desired y is the formula that fits all these data. Imagine that we have a series of measurements of voltage and current: e.g., x consists of the following pairs $(x_1^{(k)}, x_2^{(k)})$, $1 \le k \le 10$: $(1.0, 2.0), (2.0, 4.0), \ldots, (10.0, 20.0)$; we want to find a formula that is consistent with these experiments (e.g., y is the formula $x_2 = 2 \cdot x_1$).

Solution must be checkable. For a problem to be practically meaningful, we must have a way to check whether the proposed solution is correct. In other words, we must assume that there exists a feasible algorithm that checks $R(x, y)$ (given x and y). If no such feasible algorithm exists, then there is no criterion to decide whether we achieved a solution or not.

Solution must not be too long. Another requirement for a real-life problem is that in such problems, we usually know an *upper bound* for the length $\text{len}(y)$ of the description of y. In the above examples:

- In the *mathematical* problem, a proof must be not too huge, else it is impossible to check whether it is a proof or not.
- In the *physical* problem, it makes no sense to have a formula $x_2 = f(x_1, C_1, \ldots, C_{40})$ with, say, 40 parameters to describe the results $(x_1^{(1)}, x_2^{(1)}), \ldots, (x_1^{(10)}, x_2^{(10)})$ of 10 experiments, for two reasons:
  - First, one of the goals of physics is to discover the laws of nature. If the number of parameters exceeds the number of experimental data, then no matter what dependency $f(x_1, C_1, \ldots)$ we choose, in order to determine C_i, we have, say, 10 equations with 40 unknowns. Such under-determined system usually has a solution, so the fact that, say, a linear formula with many parameters fits all the experimental data does not mean that the dependency is proven to be linear: a quadratic or cubic formula with as many parameters will fit the same data as well.
  - Second, another goal of physics (definitely related to the first one) is to find a way to *compress* the data, so that we will not need to store all billions of experimental results in order to make predictions. A dependency y that requires more storage space than the original data x is clearly not satisfying this goal.

In all cases, it is necessary for a user to be able to read the desired solution symbol-after-symbol, and the time required for that reading must be feasible.

In the previous section, we have formalized "feasible time" as a time that is bounded by some polynomial of $\text{len}(x)$. The reading time is proportional to the length $\text{len}(y)$ of the answer y. Therefore, the fact the reading time is bounded by a polynomial of $\text{len}(x)$ means that the length of the output y is also bounded by some polynomial of $\text{len}(x)$, i.e., that $\text{len}(y) \le P_L(\text{len}(x))$ for some polynomial P_L.

So, we arrive at the following formulation of a problem:

Definition 2.2. *By a general practical problem (or simply a problem, for short), we mean a pair $\langle R, P_L\rangle$, where $R(x, y)$ is a feasible algorithm that transforms two binary sequences into a Boolean value ("true" or "false"), and P_L is a polynomial.*

Definition 2.3. *By an instance of a (general) problem $\langle R, P_L\rangle$, we mean the following problem:*

GIVEN: a binary sequence x.

GENERATE

- *either y such that $R(x, y)$ is true and $\text{len}(y) \le P_L(\text{len}(x))$,*
- *or, if such a y does not exist, a message saying that there are no solutions.*

For example, for the general mathematical problem described above, an instance would be: given a statement, find its proof or disproof.

Comments. What we called "general practical problems" is usually described as "problems from the class NP" (to separate them from more complicated problems in which the solution may not be easily verifiable). Problems for which there is a feasible algorithm that solves all instances are called *tractable*, *easily solvable*, or "problems from the class P" (P from *P*olynomial). It is widely believed that not all (general practical) problems are easily solvable (i.e., that NP≠P), but it has never been proved.

One way to solve an NP problem is to check $R(x, y)$ for all binary sequences y with $\text{len}(y) \le P_L(\text{len}(x))$. This algorithm (called *British Museum* algorithm) requires $2^{P_L(\text{len}(x))}$ checks. This algorithm takes exponential time and is therefore, not feasible.

2.2.3. *Reducing a Problem to Another One*

Example. Let us start with an example. Suppose that we can have an algorithm that checks whether a given system of linear inequalities

$$a_{i1} \cdot x_1 + \ldots + a_{im} \cdot x_m \geq b_i, \quad 1 \leq i \leq n,$$

with known a_{ij} and b_i, has a solution. A problem of checking whether a given system of inequalities *and equalities* $c_{k1} \cdot x_1 + \ldots + c_{km} \cdot x_m = d_k$ is consistent can be *reduced* to the problem of checking inequalities if we replace each equality by two inequalities: $c_{k1} \cdot x_1 + \ldots + c_{km} \cdot x_m \geq d_k$ and $(-c_{k1}) \cdot x_1 + \ldots + (-c_{km}) \cdot x_m \geq -d_j$ (the latter being equivalent to $c_{k1} \cdot x_1 + \ldots + c_{km} \cdot x_m \leq d_k$).

General definition. In general, we can say that a problem $\mathcal{P} = \langle R, P_L \rangle$ can be *reduced* to a problem $\mathcal{P}' = \langle R', P'_L \rangle$ if there exist three feasible algorithms U_1, U_2, and U_3 with the following properties:

- The (feasible) algorithm U_1 transforms each input x of the first problem into an input of the second problem.
- The (feasible) algorithm U_2 transforms each solution y of the first problem into the solution of the corresponding case of the second problem: i.e., if $R(x, y)$ is true, then $R'(U_1(x), U_2(y))$ is also true.
- The (feasible) algorithm U_3 transforms each solution y' of the corresponding instance of the second problem into the solution of the first problem: i.e., if $R'(U_1(x), y')$ is true, then $R(x, U_3(y'))$ is also true.

(In the above example, U_1 transforms each equality into two inequalities, and U_2 and U_3 simply do not change the values x_i at all.)

If there exists a reduction, then an instance x of the first problem is solvable if and only if the corresponding instance $U_1(x)$ of the second problem is solvable. Moreover, if we can actually solve the second instance (and find a solution y'), we will then be able to find a solution to the original instance x of the first problem (as $U_3(y')$). Thus, if we have a *feasible* algorithm for solving the second problem, we would thus design a *feasible* algorithm for solving the first problem as well.

Comment. We only described the simplest way of reducing one problem to another one: when a *single* instance of the first problem is reduced to a *single* instance of the second problem. In some cases, we cannot reduce to a *single*

case, but we can reduce to *several* cases, solving which helps us solve the original instance of the first problem.

2.2.4. *When is a Problem Tractable, and When is It Intractable?*

Definition 2.4.

- *A problem (not necessarily from the class NP) is called NP-hard if every problem from the class NP can be reduced to it.*
- *If a problem from the class NP is NP-hard, it is called NP-complete.*

If a problem $\mathcal{P}$ is NP-hard, then every feasible algorithm for solving *this* problem $\mathcal{P}$ would lead to feasible algorithms for solving *all* problems from the class NP, and this is generally believed to be hardly possible.

- For example, mathematicians believe that not only there is *no algorithm* for checking whether a give statement is provable or not (the famous Gödel's theorem has proven that), but also they believe that there is *no feasible way* to find a proof of a given statement even if we restrict the lengths of possible proofs. (In other words, mathematicians believe that computers cannot completely replace them.)
- Similarly, physicists believe that what they are doing cannot be completely replaced by computers.

In view of this belief, NP-hard problems are also called *intractable*.

Comment. It should be noted that although most scientists *believe* that intractable problems are not feasible, we still *cannot prove* (or disprove) this fact. If a NP-hard problem *can* be solved by a feasible algorithm, then (by definition of NP-hardness) *all* problems from the class NP will be solvable by feasible algorithms and thus, P=NP. Vice versa, if P=NP, then all problems from the class NP (including all NP-complete problems) can be solved by polynomial-time (feasible) algorithms.

So, if P≠NP (which is a common belief), then the fact that the problem is NP-hard means that *no matter what algorithm we use, there will always be some*

cases for which the running time grows faster than any polynomial. Therefore, for these cases, the problem is truly intractable.

2.3. Three Examples of NP-Hard Problems

Examples are needed. In this book, we will prove that some problems of data processing and interval computations are NP-hard. A typical way of proving that a certain problem $\mathcal{P}'$ is NP-hard is to reduce some problem $\mathcal{P}$, that is already known to be NP-hard, to $\mathcal{P}'$.

> Why the existence of such a reduction proves NP-hardness of $\mathcal{P}'$ is easy to explain: Since we already know that $\mathcal{P}$ is NP-hard, this means that *any* NP problem $\mathcal{P}''$ can be reduced to $\mathcal{P}$. Since $\mathcal{P}$, in its turn, is reducible to $\mathcal{P}'$, thus, $\mathcal{P}''$ can be reduced to $\mathcal{P}'$ as well. Thus, every problem from NP can be reduced to $\mathcal{P}'$ and hence, by definition, $\mathcal{P}'$ is NP-hard.

In view of this, it is very important to have *examples* of problems that are already known to be NP-hard, problems that we will use in our proofs. Many examples of such problems can be found in Garey *et al.* [120]. In this book, we will mainly use the following three NP-complete problems:

First example: Satisfiability. Historically the NP-complete problem proved to be NP-complete was the so-called *propositional satisfiability* (*3-SAT*) problem for 3−CNF formulas.

This problem consists of the following: Suppose that an integer v is fixed, and a formula F of the type $F_1 \& F_2 \& \ldots \& F_k$ is given, where each of the expressions F_j has the form $a \vee b$ or $a \vee b \vee c$, and a, b, c are either the variables $z_1, \ldots, z_v$, or their negations $\neg z_1, \ldots, \neg z_v$ (these $a, b, c, \ldots$ are called *literals*).

> For *example*, we can take a formula $(z_1 \vee \neg z_2) \& (\neg z_1 \vee z_2 \vee \neg z_3)$.

If we assign arbitrary Boolean values ("true" or "false") to v variables $z_1, \ldots, z_v$, then, applying the standard logical rules, we get the truth value of F. We say that a formula F is *satisfiable* if there exist truth values $z_1, \ldots, z_v$ for which the truth value of the expression F is "true". The problem is: given F, check whether it is satisfiable.

Second example: Partition. In the *PARTITION* problem, given n integers $s_1, \ldots, s_n$, we must check whether there exist values $x_1, \ldots, x_n \in \{-1, 1\}$ for which $s_1 \cdot x_1 + \ldots + s_n \cdot x_n = 0$.

Third example: Max-cut problem. (*MAX-CUT*) For every graph $(\mathcal{V}, \mathcal{E})$ with vertices (nodes) $\mathcal{V}$ and edges $\mathcal{E}$, and for every subset S of the set of all vertices, the *cut* $c(S)$ is defined as the number of edges from $\mathcal{E}$ that connect vertices from the set S with vertices that are outside the set S. The problem is: given a graph $(\mathcal{V}, \mathcal{E})$ and a positive integer L, check whether there exists a set $S \subseteq \mathcal{V}$ with the cut $c(S) \geq L$.