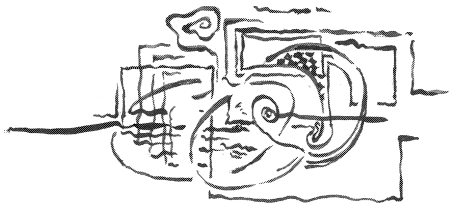

Discuss, commit errors, make mistakes,
but for God's sake think –
even if you should be wrong –
but think your own thoughts.

Gotthold Ephraim Lessing

Chapter 4

Limits of Computability or Why Do There Exist Tasks That Cannot be Solved Automatically by Computers

4.1 Aim

In Chapter 3 we discovered that there exist different infinite sizes. For instance, the number of real numbers is a larger infinity than the number of natural numbers. A infinite set is exactly as large as $\mathbb{N}$ if one can number the elements of A as the first one, the second one, the third one, etc. Here we aim to show that computing tasks exist that cannot be solved by any algorithm. The idea of our argument is simple. We show that the number of different tasks (computing problems) is a larger infinity than the number of all programs. Hence, there exist problems that cannot be algo-

rithmically solved and so their solution cannot be automatically found by means of computers. But it is not satisfactory to prove the existence of algorithmically unsolvable problems. One could think that all algorithmically unsolvable problems are so artificial that none of them is really interesting for us. Therefore, we strive to show that there are concrete problems of serious interest in practice that cannot be algorithmically solved.

This chapter is the hardest one of this book, and so you do not worry or even be frustrated when you do not get a complete understanding of all details. Many of the master students at universities do not master this topic in detail. It is already valuable if one is able to understand and correctly interpret the computer science discoveries presented in what follows. To get the full understanding of the way in which these results were discovered usually requires multiple reading and discussions of the proof ideas. How many confrontations with this hard topic you perform is up to you.

It is important to know that one can successfully study the topics of all following chapters even in the case when one did not get all the arguments of Chapter 4.

4.2 How Many Programs Do Exist?

How many programs do we have? The first simple answer is: "Infinitely many." Clearly, for each program A, there is another program B that is longer by a row (by an instruction) than A. Hence, there are infinitely many program lengths and so infinitely many programs must exist. The question of our main interest is, whether the number of programs is equal to $|\mathbb{N}|$ or not. First, we aim to show that the number of different programs is the same infinite size as the number of natural numbers. We show it by giving a numbering of programs.

Let us start with thinking about the number of texts that can be written by a computer or a typewriter. Each text can be viewed as a sequence of **symbols** of the keyboard used. We have to take

into account all small and large letters of the latin alphabet. Additionally, one is allowed to use symbols such as

?, !, ·, $, /, +, *, etc.

Moreover, every keyboard contains a key for the character blank. For instance, we use blank to separate two words or two sentences. We often use the symbol ␣ to indicate the occurrence of the character blank. Since blank has its meaning in texts, we consider it as a symbol (letter). From this point of view, texts are not only words such as

"computer" or "mother"

and not only sentences such as

"Computer ␣science ␣is ␣full ␣of ␣magic",

but also sequences of keyboard characters without any meaning such as

xyz*-+?!abe/

This means that we do not expect any meaning of a text. Semantics does not play any role in our definition of the notion of a "text". A text is simply a sequence of symbols that does not need to have any interpretation. In computer science the set of symbols used is called an **"alphabet"** and we speak about **texts over an alphabet** if all texts considered consist of symbols of this alphabet only.

Because blank is considered as a symbol, the content of any book may be viewed as a text. Hence, we can fix the following:

Every text is finite, but there is no upper bound on the length of a text. Therefore, there are infinitely many texts.

Let us observe the similarity to natural numbers. Each natural number has a finite decimal representation as a sequence of digits (symbols) $0, 1, 2, 3, 4, 5, 6, 7, 8,$ and 9. The length of the decimal

representation grows with the number, and so there is no bound on the representation length of natural numbers[1].

It looks like that one can conjecture that

> *"The number of all texts over the characters of a keyboard is equal to* $|\mathbb{N}|$*."*

This is really true and we prove it by enumerating the texts. It is sufficient to show that one can order all texts in an infinite list. The ordering works in a similar way to creating a dictionary, but not exactly in the same way. Following the sorting rules of the dictionary, we have to take first the texts $a, aa, aaa, aaaa$, etc., and we will never order texts containing a symbol different from a, because there are infinitely many texts consisting of the letter a only. Therefore, we have to change the sorting approach a little bit. To order all texts in a list, we first apply the following rule:

> *Shorter texts are always before longer texts.*

This means that our infinite list of all texts starts with all texts of the length 1, then the texts of the length 2 follow, after then the texts of the length 3, etc. What still remains is to fix the order of the texts of the same length for any length. If one uses the letters of the latin alphabet only, then one can do it in the same way as used in dictionaries. This means to start with texts that begin with the letter a, etc. Since we also have a lot of special symbols on our keyboard, such as ?, !, *, +, etc., we have to order the symbols of our keyboard alphabet first. Which order of symbols we take is our choice and for our claim about the cardinality of the set of all texts over the keyboard alphabet it does not matter. Fig. 4.1 depicts a possible ordering of the symbols of the keyboard alphabet. Having an order of the alphabet symbols

> *one sorts the texts of the same length in the same way as in dictionaries*[2]*.*

[1] This means that one cannot restrict the representation length by a concrete number. For instance, if one upperbounds the representation length by n, then one would have at most 10^n different representation available and this is not enough to represent all infinitely many natural numbers.

[2] Usually, we speak about the lexicographical order.

1	2	3	...	25	26	27	28	...	51	52	53	54	...	61	62
a	b	c	...	y	z	A	B	...	Y	Z	1	2	...	9	0

63	64	65	66	67	68	69	70	71	72	73	74	75	...	167
+	”	*	ç	&	!	.	:	,	;	?	$	£	...	␣

Fig. 4.1

This means that for the same text length, we start with texts beginning with the first symbol of our alphabet ordering. For instance, taking the order of symbols depicted in Fig. 4.1, the numbering of texts starts with

1 a
2 b
3 c
⋮
167 ␣
⋮

Then, the texts of length 5 are ordered as follows:

aaaaa
aaaab
aaaac
⋮
aaaa␣
aaaba
aaabb
aaabc
⋮

Why did we take time to show that the number of texts is equal to $|\mathbb{N}|$? Because

each program is a text over the keyboard alphabet.

Programs are nothing else than special texts that are understandable for computers. Therefore, the number of all programs is not larger than the number of all texts over the keyboard alphabet and so we can claim

"The number of programs is equal to $|\mathbb{N}|$."

What we really have showed is that the number of programs is infinite and not larger than $|\mathbb{N}|$. The equality between $|\mathbb{N}|$. and the number of programs is the consequence of the fact that $|\mathbb{N}|$ is the smallest infinite size. But we did not prove this fact. Hence, if we want to convince the reader and provide a full argumentation of this fact, then we have to find a matching between $\mathbb{N}$ and programs. As we already know, any numbering provides a matching. And

one gets a numbering of all programs by erasing all texts that do not represent any program from our infinite list of all texts over the keyboard alphabet.

It is important to observe that the deletion of texts without an interpretation as programs can even be done automatically. One can write programs, called **compiles**, that get texts as inputs and decide whether a given text is a program in the programming language considered or not. It is worth being to note that

a compiler can check the syntactical correctness of a text as a program but not the semantical correctness.

This means that a compiler checks whether a text is a correctly written sequence of computer instructions, i.e., whether the text is a program. A compiler does not verify, whether the program is an algorithm, i.e., whether the program does something reasonable or not, or whether the program can repeat a loop infinitely many times.

Hence, we are allowed to number all programs and so to list all programs in a sequence

$P_0, P_1, P_2, P_3, \ldots, P_i, \ldots$

where $\boldsymbol{P_i}$ denotes the **i-th program**.

Why is it important for us that the number of programs and so the number of algorithms is not larger than $|\mathbb{N}|$. The answer is that the number of all possible computing tasks is larger than $|\mathbb{N}|$ and so there are more problems than algorithms. The immediate consequence is that there exist problems that cannot be solved by any algorithms (for which no solution method exists).

Allusively, we have already showed in Chapter 3 that the number of problems is very large. For each real number c, one can consider the following computer task Problem(c).

Problem(c)

Input: a natural number n

Output: a number c up to n decimal digits after the decimal point

We say that an **algorithm A_c solves Problem(c)** or that **A_c generates c**, if, for any given $n \in \mathbb{N}$, A_c outputs all digits of c before the decimal point and the first n digits of c after the decimal point.

For instance,

- for $c = \frac{4}{3}$ and an input $n = 5$, the algorithm $A_{\frac{4}{3}}$ has to give the output 1.33333.
- for $\sqrt{2}$, the algorithm $A_{\sqrt{2}}$ has to generate the number 1.4142 for the input $n = 4$ and the number 1.414213 for the input $n = 6$.
- for $c = \pi$, the algorithm A_π has to provide the output 3.141592 for $n = 6$.

Exercise 4.1 a) What is the output of an algorithm $A_{\frac{17}{6}}$ generating 17/6 fort he input $n = 12$?

b) What are the outputs of an algorithm A_π that generates π for inputs $n = 2, n = 0, n = 7$, and $n = 9$?

Exercise 4.2 (challenge) Can you present a method fort he generation of π up to an arbitrary large number of digits after the decimal point?

In Chapter 3, we proved that the number of real numbers is larger than $|\mathbb{N}|$, i.e., that $|\mathbb{R}| > |\mathbb{N}|$. Since the number of algorithms is not larger than $|\mathbb{N}|$, the number of real numbers is larger than the number of algorithms. Therefore, we can conclude that

> *there exists a real number c such that Problem(c) is not algorithmically solvable.*

Thus, we proved that there are real numbers that cannot be generated by any algorithm. Do we understand exactly, what does it mean? Let us try to build our intuition in order to get a better understanding of this result. The objects as natural numbers, rational numbers, texts, programs, recipes, and algorithms have something in common.

> *All these objects have a finite representation.*

But his is not true for real numbers. If one can represent a real number in a finite way, then one can view this representation as a text. Since the number of different texts is smaller than the number of real numbers, there must exist a real number without any finite representation.

What does it exactly mean? To have a constructive description of a real number e means that one is able to generate e completely digit by digit. Also if the number e has an infinite decimal representation, one can use the description to unambiguously estimate the digit on any position of its decimal representation. In this sense, the finite description of e is complete. In other words, such a finite description of e provides an algorithm for generating e. For instance, $\sqrt{2}$ is a finite description of the irrational number $e = \sqrt{2}$ and we can compute this number with an arbitrarily high precision by an algorithm[3]. Therefore, we are allowed to say:

> *Real numbers having a finite representation are exactly the numbers that can be algorithmically generated and there exist real numbers that do not possess a finite representation and so are not computable (algorithmically generable).*

[3]For instance, by the algorithm of Heron.

Exercise 4.3 What do you mean? Are there more real numbers with finite representations than real numbers without any finite representation, or vice versa? Justify your answer!

We see that there are tasks that cannot be solved by algorithms. But we are not satisfied with this knowledge. Who is interested in asking for an algorithm generating a number e that does not have any finite representation? How can one formulate such a task in a finite way? Moreover, when only tasks of this kind are not algorithmically solvable, then we are happy and foract about this "artificial" theory and dedicate our time to solving problems of a practical relevance. Hence, you may see the reason why we do not stop with our investigation here and are not content with our achievements. We have to continue in our study in order to fix, whether there are interesting computing tasks with a finite description that cannot be automatically solved by means of computers.

4.3 YES or NO Is the Question or Another Application of Diagonalization

The simplest problems considered in computer science are probably decision problems. A decision problem is to recognize whether a given object has a special property we are searchin for or not. For instance, one gets a digital picture and has to decide whether a chair is on the picture. One can also ask whether a person is on the picture or even whether a specific person (for instance, Albert Einstein) is on the picture. The answer has to be unambiguous "YES" or "NO". Other answers are not allowed and we force that the answer is always correct.

Here, we consider a simple kind of decision problems. Let M be an arbitrary subset of $\mathbb{N}$, i.e., let M be a set that contains some natural numbers. We specify the **decision problem** $(\mathbb{N}, M)$ as follows.

Input: a natural number n from $\mathbb{N}$

Output:

"YES" if $\boldsymbol{n}$ belongs to $\boldsymbol{M}$
"NO" if $\boldsymbol{n}$ does not belong to $\boldsymbol{M}$

For instance, one can take PRIME as $\boldsymbol{M}$, where

$$\text{PRIME} = \boldsymbol{\{2, 3, 5, 7, 11, 13, 17, 19, \ldots\}}$$

is the finite set of all primes. Then, (ℕ, PRIME) ist he problem to decide whether a given natural number $\boldsymbol{n}$ is a prime or not. The problem $\boldsymbol{(\mathbb{N}, \mathbb{N}_{even})}$ is to decide whether a given nonnegative integer is even or not.

For each subset $\boldsymbol{M}$ of ℕ we say that an **algorithm $\boldsymbol{A}$ recognizes $\boldsymbol{M}$** ort hat an **algorithm $\boldsymbol{A}$ solves the decision problem $\boldsymbol{(\mathbb{N}, M)}$**, if, for any input n, A computes

(i) the answer "YES" if n belongs to M, and

(ii)the answer "NO" if n does not belong to M ($n \notin M$).

Sometimes one uses the digit "1" instead of "YES" and the digit "0" instead of "NO". If A answers "YES" for an input n, then we say that **algorithm $\boldsymbol{A}$ accepts the number $\boldsymbol{n}$**.If A outputs "NO" for an input n, then we say that **algorithm $\boldsymbol{A}$ rejects the number $\boldsymbol{n}$**.Thus, an algorithm recognizing PRIME, accepts each prime and rejects each composite number.

If there exists an algorithm solving a decision problem $(\mathbb{N}, M)$, then we say that the problem $(\mathbb{N}, M)$ is **algorithmically solvable** or that

the problem $\boldsymbol{(\mathbb{N}, M)}$ is decidable.

Clearly, the decision problem $(\mathbb{N}, \mathbb{N}_{even})$ is decidable. It is sufficient to verify whether a given natural number is even or odd. The problem (ℕ, PRIME) is also decidable because we know how to check whether a natural number is a prime or not and it is not too complicated to describe such a method as an algorithm.

Exercise 4.4 The naive method for primality testing is to divide the given number n by all numbers between 2 and $n-1$. If none of these $n-2$ numbers divides

n, then n is a prime. To test primality in this way means to perform a lot of work. For the number 1000002 one has to execute 1000000 divisibility tests. Can you propose another method that can verify primality by performing an essentially smaller number of divisibility tests?

Exercise 4.5 (challenge) Write a program in the programming language TRANSPARENT of Chapter 2 that solves the problem (ℕ, QUAD) where

$$\text{QUAD} = \{1, 4, 9, 16, 25, \ldots\}$$

is the set of all squares i^2.

First, we aim to show the existence of decision problems that are not algorithmically solvable. Such decision problems are called

undecidable or **algorithmically unsolvable**.

We already recognized that we can list all programs as $P_0, P_1, P_2, \ldots$ and later we will see that one can do it by an algorithm. To list all algorithms by an algorithm is not so easy. Therefore, we begin our effort by proving a stronger result than we really need. We show that there are decision problems that cannot be solved by any program. What does "solved by a program" mean? What is the difference between algorithmic solvability and solvability by a program?

Remember. Each algorithm can be written as a program, but it does not hold that each program is an algorithm. A program can perform a pointless work. A program can perform an infinite work for some inputs without producing any result. But an algorithm must always finish ist work in a *finite time* and produce *correct result.*

Let M be a subset of ℕ. We say that a **program P accepts the set P**, if, for any given natural number n

(i) "YES", if n belongs to M, and

(ii)P outputs "NO" or works **infinitely long** if n does not belong to M.

For a program P, $\boldsymbol{M(P)}$denotes the set M accepted by P. In this way, P can be viewed as a finite representation of the potentially infinite set $M(P)$.

Immediately, we see the difference between the recognition of M by an algorithm and the acceptance of M by a program. For inputs from M both the algorithm and the program are required to work correctly and provide the right answer "YES" in a finite time (see the requirement (i)). In contrast to an algorithm, for numbers not belonging to M, a program is allowed to work infinitely long and so never produce any answer. In this sense algorithms are special programs that never run infinite computations. Therefore, it is sufficient to show that there is no program accepting a set M and the direct consequence is that there does not exist any algorithm that recognizes M (i.e., solves the decision problem $(\mathbb{N}, M)$.

To construct such a "hard" subset M of $\mathbb{N}$, we use the diagonalization method from Chapter 3 again. To this purpose, we need the following infinite representation of subsets of natural numbers (Fig. 4.2).

	0	1	2	3	4	...	i	$i+1$	...
M	0	1	0	0	1	...	1	0	...

Fig. 4.2

M is represented as an infinite sequence of bits. The sequence starts with the position 0 and has 1 at the i-th position if and only if the number i is in M. If i is not in M, then the bit 0 is on the i-th position of the sequence. The set M in Fig. 4.2 contains the numbers $1, 4$, and i. The numbers $0, 2, 3$, and $i+1$ are not in M. The binary representation of $\mathbb{N}_{even}$ looks as follows

101010101010101010 ...

The representation of PRIM starts with the following bit sequence:

0011010100010100 ...

Exercise 4.6 Write the first 17 bits of the binary representation of QUAD.

Now, we again build a two-dimensional table that is infinite in both directions. The columns are given by the infinite sequence of all numbers

$$0, 1, 2, 3, 4, 5, \ldots, i, \ldots.$$

The rows are given by the infinite sequence of all programs

$$P_0,\ P_1,\ P_2,\ P_3,\ \ldots,\ P_i,\ \ldots$$

that reads an input number only once and their only possible outputs are "YES" and "NO". One can recognize such programs by looking whether they contain only one instruction of reading and whether the only output instructions are writing the text "YES" or the text "NO". Each such program P_i unambiguously defines a set $M(P_i)$ of all natural numbers that are accepted[4] by P_i. Those numbers, for which P_i outputs "NO" or works infinitely long, do not belong to $M(P_i)$.

The rows of our table are the binary representations of sets $M(P_i)$. The j-th row (see Fig. 4.3) contains the binary representation of the set $M(P_i)$ that is accepted by the program P_j. The intersection of the i-th row and the j-th column contains "1" if P_i accepts the number j (if P_i halts on the input j with the output "YES"). The symbol "0" lies in the intersection of the i-th row and the j-th column, if P_i outputs "NO" or works infinitely long for the input j. Hence

> *the infinite table contains in its rows the representation of* ***all*** *subsets of* $\mathbb{N}$ *that can be accepted by a program.*

Next we aim to show that there is at least one subset of $\mathbb{N}$ in the table, i.e., that there is a subset of $\mathbb{N}$, whose binary representation differs from each row of the table (Fig. 4.3). We show it by constructing a sequence of bits, called DIAG, that does not occur in any row of the table. The construction of the bit sequence DIAG and the corresponding set $M(\text{DIAG})$ is done by the diagonalization method.

[4] P_i finishes the work on them with printing "YES".

	0	1	2	3	4	5	6	$\cdots$	i	$\cdots$	j	$\cdots$
$M(P_0)$	**0**	1	1	0	0	1	0		1		0	
$M(P_1)$	0	**1**	0	0	0	1	1		0		0	
$M(P_2)$	1	1	**1**	0	0	1	0		1		1	
$M(P_3)$	1	0	1	**0**	1	0	1		1		0	
$M(P_4)$	0	0	0	1	**1**	0	1		0		1	
$M(P_5)$	1	1	1	1	1	**1**	1		1		1	
$M(P_6)$	1	0	1	0	0	0	**1**		0		1	
$\vdots$												$\cdots$
$M(P_i)$	0	1	1	0	0	1	0		**1**			
$\vdots$												$\cdots$
$M(P_j)$	1	0	1	0	1	1	1				**0**	
$\vdots$												$\vdots$

Fig. 4.3

	0	1	2	3	4	5	6	$\cdots$	i	$\cdots$	j	$\cdots$
DIAG	1	0	0	1	0	0	0		0		1	$\cdots$

Fig. 4.4

First, see the binary value a_{00} in the intersection of the 0-th row and the 0-th column. If $a_{00} = 0$ (Fig. 4.3), i.e, if 0 does not belong to $M(P_i)$, then we set the 0-th position d_0 of DIAG to 1 (i.e., if 0 is in $M(P_1)$), then we set $d_0 = 0$ (i.e., we do not take o to M(DIAG)). After this first step pf the construction of DIAG we fixed only the value of the first position of DIAG and due to this we are sure that DIAG differs from the 0-th row if the table (i.e., from $M(P_0)$) at least in the 0-th element.

Analogously, we continue in the second construction step. We consider the second diagonal square, where the first row intersects the first column. We aim to choose the first position d_1 of DIAG is such a way that DIAG differs from the binary representation of $M(P_1)$ at least in the value of position. Therefore, if $a_{11} = 1$ (i.e., if 1 is $M(P_1)$), we set d_1 to 0 (i.e., we do not take 1 into M(DIAG)). If $a_{11} = 0$ (i.e., if 1 is not in $M(P_1)$), then we set $d_1 = 1$ (i.e., we take 1 into M(DIAG)).

If $\bar{a}_{ij}$ represents the opposite value to a_{ij} for any bit in the intersection of the i-th row and the j-th column (the opposite value to 1 is $\bar{1} = 0$ and $\bar{0} = 1$ is the opposite value to 0), then, after two construction steps, we reached the situation as depicted in Fig. 4.5

	0	1	2	3	4	$\cdots$	i	$i+1$	$\cdots$
DIAG	$\bar{a}_{00}$	$\bar{a}_{11}$	?	?	?	$\cdots$	?	?	$\cdots$

Fig. 4.5

The first two elements of DIAG are $\bar{a}_{00}$ and $\bar{a}_{11}$, and so DIAG differs from both $M(P_0)$ and $M(P_1)$. The remaining positions of DIAG are still not determined we aim to fix them in such a way that DIAG will differ from each row of the table in Fig. 4.3.

In general, we guarantee a difference between DIAG and the i-th row of the table in Fig. 4.3 as follows. Remember that $\bar{a}_{ii}$ is the bit of the square in the intersection of the i-th row and the i-th column and that d_i denotes the i-th bit of DIAG. If $\bar{a}_{ii} = 1$ (i.e., if i belongs to $M(P_i)$), then we set $d_i = 1$ (i.e., we do not take i into $M(\text{DIAG})$). If $\bar{a}_{ii} = 0$ (i.e., if i is not in $M(P_i)$), then we set $d_i = 1$ (i.e., we take i into $M(\text{DIAG})$). Hence, $M(\text{DIAG})$ differs from $M(P_i)$.

In this way DIAG is constructed in such a way that it does not occur in any row of the table. For a concrete, hypothetical table in Fig. 4.3, Fig. 4.4 shows the corresponding representation of DIAG. In general, one can outline the representation of DIAG as done in Fig. 4.6.

	0	1	2	3	4	$\cdots$	i	$\cdots$
DIAG	$\bar{a}_{00}$	$\bar{a}_{11}$	$\bar{a}_{22}$	$\bar{a}_{33}$	$\bar{a}_{44}$	$\cdots$	$\bar{a}_{ii}$	$\cdots$

Fig. 4.6

In this way we obtain that

M(DIAG) is not accepted by any program and therefore, the decision problem ($\mathbb{N}$, M(DIAG)) cannot be solved by any algorithm.

One can specify M(DIAG) also in the following short way:

$$\begin{aligned} M(\text{DIAG}) &= \{n \in \mathbb{N} \mid n \text{ is not in } M(P_n)\} \\ &= \text{the set of all natural numbers } n, \\ &\quad \text{such that } n \text{ is not in } M(P_n). \end{aligned}$$

Exercise 4.7 Assume, the intersection of the first 10 rows and the first 10 columns in the table of all programs contains values as written in Fig. 4.7. Estimate the first 10 positions of DIAG.

	0	1	2	3	4	5	6	7	8	9	$\cdots$
$M(P_0)$	1	1	1	0	0	1	0	1	0	1	
$M(P_1)$	0	0	0	0	0	0	0	0	0	0	
$M(P_2)$	0	1	1	0	1	0	1	1	0	0	
$M(P_3)$	1	1	1	0	1	1	0	0	0	0	
$M(P_4)$	1	1	1	1	1	1	1	0	1	0	
$M(P_5)$	0	0	1	0	0	1	0	1	1	0	
$M(P_6)$	1	0	0	0	1	0	1	0	0	0	
$M(P_7)$	1	1	1	1	1	1	1	1	1	1	
$M(P_8)$	0	0	1	1	0	0	1	1	0	0	
$M(P_9)$	1	0	1	0	1	0	1	0	1	0	
$M(P_{10})$	0	0	1	0	0	0	1	1	0	1	
$\vdots$											$\ddots$

Fig. 4.7

Exercise 4.8 (challenge) Consider that

$$M(\text{2-DIAG}) = \text{the set of all even numbers } 2i, \text{ such that } 2i \text{ is not in } M(P_i).$$

Is the decision problem ($\mathbb{N}$, M(2-DIAG)) algorithmically solvable or not? Explain carefully your argument! Draw also pictures that would similarly to Fig. 4.3 and Fig. 4.4 show the construction of 2-DIAG.

Exercise 4.9 (challenge) Can you use the solution to Exercise (4.8) in order to define two other subsets of $\mathbb{N}$ that are not algorithmically solvable? How many algorithmically unsolvable problems can be derived by diagonalization?

Exercise 4.10 (challenge) Consider

$M(\mathrm{DIAG}_2)$ as the set of all even natural numbers $2i$ such that $2i$ is not in $L(P_{2i})$.

Can you say something about the algorithmical solvability of $(\mathbb{N}, M(\mathrm{DIAG}_2))$?

Now, we know that the decision problem $(\mathbb{N}, M(\mathrm{DIAG}))$ is not algorithmically solvable. But we are not satisfied with this result. The problem looks to be described in a finite way by our construction, though it is represented by an infinite sequence of bits. But our construction does not provide any algorithm for generating DIAG, because as we will see later, though the table in Fig. 4.3 really exists, it cannot be generated by an algorithm. Moreover, the decision problem $(\mathbb{N}, M(\mathrm{DIAG}))$ does not correspond natural task arising in practice.

4.4 Die Methode der Reduktion, oder: Wie sich eine erfolgreiche Problemlösungsmethode zur Erzeugung negativer Resultate ausnutzen lässt

Wir wissen jetzt, dass wir mittels der Diagonalisierungsmethode algorithmisch unlösbare Probleme beschreiben können. Das bringt uns in eine gute Anfangsposition. In diesem Abschnitt geht es darum, wie man die Beweise der algorithmischen Unlösbarkeit geschickt auf andere Probleme ausbreiten kann. Die Idee ist, eine Relation „**leichter oder gleich schwer**“ bezüglich algorithmischer Lösbarkeit einzuführen.

Seien $\boldsymbol{U_1}$ und $\boldsymbol{U_2}$ zwei Probleme. Wir sagen

$\boldsymbol{U_1}$ ist leichter oder gleich schwer wie $\boldsymbol{U_2}$

oder

$\boldsymbol{U_2}$ ist nicht leichter als $\boldsymbol{U_1}$

bezüglich algorithmischer Lösbarkeit und schreiben