Alberto Pettorossi

Elements of Computability, Decidability, and Complexity

Third Edition

ARACNE

Table of Contents

Preface

These lecture notes are intended to introduce the reader to the basic notions of computability theory, decidability, and complexity. More information on these subjects can be found in classical books such as [6,8,15,16,33]. The results reported in these notes are taken from those books and in various parts we closely follow their style of presentation. The reader is encouraged to look at those books for improving his/her knowledge on these topics. Some parts of the chapter on complexity are taken from the lecture notes of a beautiful course given by Prof. Leslie Valiant at Edinburgh University, Scotland, in 1979. It was, indeed, a very stimulating and enjoyable course.

For the notions of Predicate Calculus we have used in this book the reader may refer to [26].

I would like to thank Dr. Maurizio Proietti at IASI-CNR (Roma, Italy), my colleagues, and my students at the University of Roma Tor Vergata and, in particular, Michele Martone. They have been for me a source of continuous inspiration and enthusiasm.

Finally, I would like to thank Dr. Gioacchino Onorati and Lorenzo Costantini of the Aracne Publishing Company for their helpful cooperation.

Roma, July 2006

Preface to the Second Edition.
Some typographical errors which were present in the first edition have been corrected. The chapters on Turing Machines and Decidability Theory have been revised.

Roma, July 2007

Preface to the Third Edition.
The chapter on Computational Complexity has been revised and some minor misprints have been corrected.

Roma, October 2009

Alberto Pettorossi
Department of Informatics, Systems, and Production
University of Roma Tor Vergata
Via del Politecnico 1,
I-00133 Roma, Italy
email: `adp@info.uniroma2.it`
URL: `http://www.iasi.cnr.it/~adp`

Chapter 1

Turing Machines

1 Basic Definitions

$N = \{0, 1, 2, \ldots\}$ is the set of natural numbers. In what follows we will feel free to say 'number', instead of 'natural number'.

Given a set A and a number $k \geq 0$, let A^k denote the set of all k-tuples of elements of A. Thus, by definition, we have that:
(i) A^0 is the singleton made out of the 0-tuple $\langle\rangle$ (also called *empty tuple* or *empty string* or *empty word* and denoted by ε), that is, $A^0 = \{\langle\rangle\}$,
(ii) $A^1 = \{\langle x\rangle \mid x \in A\}$, and
(iii) $A^k = \{\langle x_1, \ldots, x_k\rangle \mid x_1, \ldots, x_k \in A\}$ for any $k > 1$.
We identify $\langle x\rangle$ with x for each $x \in A$. Thus, $A^1 = A$. Sometimes a tuple $\langle x_1, \ldots, x_k\rangle$ is also denoted by $x_1 \ldots x_k$.
Let A^* denote the set of all finite k-tuples of elements of A, with $k \geq 0$, that is,
$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots \cup A^k \cup \ldots$
An element of A^* is also called a *string* or a *word*. A^* is also said to be the set of *strings* or *words over* A. The *length* of an element w of A^*, denoted $|w|$, is k iff w belongs to A^k, for any $k \geq 0$.

A set A is said to be *denumerable* (or *countable*) iff A has the cardinality of N or there exists a natural number n such that A has the same cardinality of n. The *cardinality* of a set A will be denoted by $|A|$.

We have that A is denumerable iff A^* is denumerable. The proof is based on the dove-tailing technique [32, Chapter 1].

Definition 1. [Irreflexive Total Order] Given a set A we say that a binary relation $< \subseteq A \times A$ is an *irreflexive total order* (or a *total order*, for short) on A iff (i) $<$ is a transitive relation, and (ii) for any two elements $a, b \in A$ we have that exactly one of these three cases occurs: either $a<b$, or $a=b$, or $b<a$.

Let us consider a denumerable set A of symbols with a total order $<$ among its elements. Then the *canonical order*, on the set of all words in A^* is a binary relation, also denoted $<$, which is the subset of $A^* \times A^*$ defined as follows. Given any two words w_1 and w_2 in A^*, we say that w_1 *canonically precedes* w_2, and we write $w_1 < w_2$, iff (i) *either* $|w_1|$ is smaller than $|w_2|$, (ii) *or* $|w_1| = |w_2|$ and $w_1 = p\,x\,q_1$ and $w_2 = p\,y\,q_2$ for some strings p, q_1, and q_2 in A^* and $x, y \in A$ such that $x < y$ (according to the order $<$ on A). We have that the order $<$ on A^* is transitive.

For instance, if $A = \{a, b\}$ and $a < b$ then the canonical order on A^* is the one which is denoted by the sequence: $\varepsilon < a < b < aa < ab < ba < bb < aaa < aab < aba < \ldots < bbb < \ldots < aaaa < \ldots$

According to the above definition, any total order is an irreflexive order. In the literature (see, for instance, [32, Chapter 1]) there is also a different notion of total order which is based on a partial order, and thus, it is a reflexive order. This second notion is defined as follows.

Definition 2. [Reflexive Total Order] Given a set A we say that $\leq \subseteq A \times A$ is a *reflexive total order* (or a *total order*, for short) on A iff (i) $\leq$ is reflexive, antisymmetric, and transitive (and thus, $\leq$ is a partial order), and (ii) for any two elements $a, b \in A$ we have that $a \leq b$ or $b \leq a$.

One can show that for any given total order $<$ on a set A according to Definition 1, there exists a total order $\leq$ according to Definition 2, such that for any two elements $a, b \in A$, we have that $a \leq b$ iff $a < b$ or $a = b$.

2 Turing Machines

In 1936 the English mathematician Alan Turing introduced an abstract model for computation called Turing Machine [39]. There are various variants of this model, which can be proved to be equivalent in the sense that they are all capable to compute the same set of functions from N to N.

Informally, a *Turing Machine* M consists of:
(i) a *finite automaton* FA, also called the *control*,
(ii) a one-way infinite *tape* (also called *working tape* or *storage tape*), which is an infinite sequence $\{c_i \mid i \in N, i > 0\}$ of *cells* c_i's, and
(iii) a tape head which at any given time *is on* a single cell. When the tape head is on the cell c_i we will also say that the tape head *scans* the cell c_i.

The cell which the tape head scans, is called the *scanned cell* and it can be read and written by the tape head. Each cell contains exactly one of the symbols of the *tape alphabet* Γ. The states of the automaton FA are also called *internal states*, or simply *states*, of the Turing Machine M.

We say that the Turing Machine M *is in state* q, or q is the current state of M, if the automaton FA *is in state* q, or q is the current state of FA, respectively.

We assume a left-to-right orientation of the tape by stipulating that for any $i > 0$, the cell c_{i+1} is immediately to the right of the cell c_i.

A Turing Machine M behaves as follows. It starts with a tape containing in its leftmost $n\,(\geq 0)$ cells $c_1\, c_2 \ldots c_n$ a sequence of n input symbols from the *input alphabet* Σ, while all other cells contain the symbol B, called *blank*, belonging to Γ. We assume that: $\Sigma \subseteq \Gamma - \{B\}$. If $n = 0$ then, initially, the blank symbol B is in every cell of the tape. The Turing Machine M starts with its tape head on the leftmost cell, that is, c_1, and its control, that is, the automaton FA in its initial state q_0.

An *instruction* (or a *quintuple*) of the Turing Machine is a structure of the form:

$q_i,\, X_h \longmapsto q_j,\, X_k,\, m$

where: (i) $q_i \in Q$ is the *current state* of the automaton FA,

(ii) $X_h \in \Gamma$ is the *scanned symbol*, that is, the symbol of the scanned cell that is read by the tape head,

(iii) $q_j \in Q$ is the *new state* of the automaton FA,

(iv) $X_k \in \Gamma$ is the *printed symbol*, that is, the non-blank symbol of Γ which replaces X_h on the scanned cell when the instruction is executed, and

(v) $m \in \{L, R\}$ is a value which denotes that, after the execution of the instruction, the tape head moves *either* one cell to the left, if $m = L$, *or* one cell to the right, if $m = R$. Initially and when the tape head of a Turing Machine scans the leftmost cell c_1 of the tape, m must be R. (Alternatively, as we will see below, we may assume that if m is L then the new state q_j of the automaton FA is not a final state and there are no quintuples in which q_j occurs in the first position.)

Given a Turing Machine M, if no two instructions of that machine have the same current state q_i and scanned symbol X_h, we say that the Turing Machine M is *deterministic*.

Since it is assumed that the printed symbol X_k is *not* the blank symbol B, we have that if the tape head scans a cell with a blank symbol then: (i) every symbol to the left of that cell is *not* a blank symbol, and (ii) every symbol to the right of that cell is a blank symbol.

Now we give the formal definition of a Turing Machine.

Definition 3. [Turing Machine] A *Turing Machine* is a septuple of the form $\langle Q, \Sigma, \Gamma, q_0, B, F, \delta\rangle$, where:
- Q is the set of *states*,
- Σ is the *input alphabet*,
- Γ is the *tape alphabet*,
- q_0 is the initial state,
- B is the *blank symbol*,
- F is the set of the *final states*, and
- δ is a partial function from $Q \times \Gamma$ to $Q \times (\Gamma - \{B\}) \times \{L, R\}$, called the *transition function*, which defines the set of instructions *or* quintuples of the Turing Machine.

We assume that Q and Γ are disjoint sets, $\Sigma \subseteq \Gamma - \{B\}$, $q_0 \in Q, B \in \Gamma$, and $F \subseteq Q$.

Let us consider a Turing Machine whose leftmost part of the tape consists of the cells:

$c_1\, c_2 \ldots c_{h-1}\, c_h \ldots c_k$

where c_k, with $1 \leq k$, is the rightmost cell with a non-blank symbol, and c_h, with $1 \leq h \leq k+1$, is the cell scanned by the tape head.

We may extend the definition of a Turing Machine by allowing the transition function δ to be a *partial function* from the set $Q \times \Gamma$ to the set of the subsets of $Q \times (\Gamma - \{B\}) \times \{L, R\}$ (not to the set $Q \times (\Gamma - \{B\}) \times \{L, R\}$). In that case it is possible that two quintuples of δ have the same first two components and if this is the case, we say that the Turing Machine is *nondeterministic*.

Unless otherwise specified, the Turing Machines we consider are assumed to be deterministic.

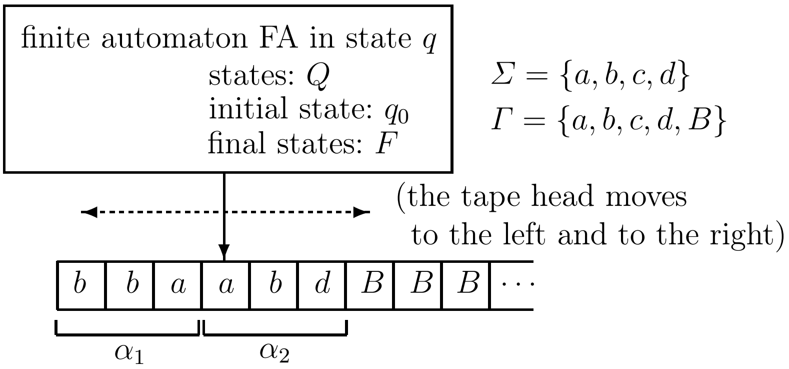

finite automaton FA in state q
states: Q
initial state: q_0
final states: F
$\Sigma = \{a, b, c, d\}$
$\Gamma = \{a, b, c, d, B\}$
(the tape head moves to the left and to the right)
b b a a b d B B B ···
α_1
α_2

Fig. 1. A Turing Machine in the configuration $\alpha_1\, q\, \alpha_2$, that is, $b\, b\, a\, q\, a\, b\, d$. The state $q \in Q$ is the current state of the Turing Machine. The head scans the cell c_4 and reads the symbol a.

Definition 4. [Configuration of a Turing Machine] A *configuration* of a Turing Machine M whose tape head scans the cell c_h for some $h \geq 1$, such that the cells containing a non-blank symbol in Γ are $c_1 \ldots c_k$, for some $k \geq 0$, with $1 \leq h \leq k{+}1$, is the triple $\alpha_1\, q\, \alpha_2$, where:

- α_1 is the (possibly empty) word in $(\Gamma - \{B\})^{h-1}$ written in the cells $c_1\, c_2 \ldots c_{h-1}$, one symbol per cell from left to right,

- q is the current state of the Turing Machine M, and

- if the scanned cell c_h contains a non-blank symbol, that is, $1 \leq h \leq k$, for some $k \geq 1$, then α_2 is the non-empty word of Γ^{k-h+1} written in the cells $c_h \ldots c_k$, one symbol per cell from left to right, else if the scanned cell contains the blank symbol B, then α_2 is the sequence of one B only, that is, $h = k{+}1$, for some $k \geq 0$ (see also Figure 1).

For each configuration $\gamma = \alpha_1\, q\, \alpha_2$, we assume that: (i) the tape head scans the leftmost symbol of α_2, and (ii) we say that q is *the state in* the configuration γ.

If the word $w = a_1 a_2 \ldots a_n$, for $n \geq 0$, is initially written, one symbol per cell, on the n leftmost cells of the tape of a Turing Machine M and all other cells contain B, then the *initial configuration* of M is $q_0\, w$, that is, the configuration where: (i) α_1 is the empty sequence ε, (ii) the state of M is the initial state q_0, and (iii) $\alpha_2 = w$. The word w of the initial configuration is said to be the *input word* for the Turing Machine M.

Definition 5. [Tape Configuration of a Turing Machine] Given a Turing Machine whose configuration is $\alpha_1\, q\, \alpha_2$, we say that its *tape configuration* is the string $\alpha_1\, \alpha_2$ in Γ^*.

Sometimes, by abuse of language, instead of saying 'tape configuration', we will simply say 'configuration'. The context will tell the reader whether the intended meaning of the word 'configuration' is that of Definition 4 or that of Definition 5.

Now we give the definition of a move of a Turing Machine. By this notion we characterize the execution of an instruction as a pair of configurations, that is, (i) the configuration 'before the execution' of the instruction, and (ii) the configuration 'after the execution' of the instruction.

Definition 6. [Move (or Transition) of a Turing Machine] Given a Turing Machine M, its *move relation* (or *transition relation*), denoted $\rightarrow_M$, is a subset of $C_M \times C_M$, where C_M is the set of configurations of M, such that for any state $p, q \in Q$, for any tape symbol $X_1, \ldots, X_{i-2}, X_{i-1}, X_i, X_{i+1}, \ldots, X_n, Y \in \mathit{\Gamma}$, either:

1. if $\delta(q, X_i) = \langle p, Y, L\rangle$ and $X_1 \ldots X_{i-2}X_{i-1} \neq \varepsilon$ then
$X_1 \ldots X_{i-2}X_{i-1}\, q\, X_i X_{i+1} \ldots X_n \;\rightarrow_M\; X_1 \ldots X_{i-2}\, p\, X_{i-1}\, Y\, X_{i+1} \ldots X_n$

or

2. if $\delta(q, X_i) = \langle p, Y, R\rangle$ then
$X_1 \ldots X_{i-2}X_{i-1}\, q\, X_i X_{i+1} \ldots X_n \;\rightarrow_M\; X_1 \ldots X_{i-2}X_{i-1}\, Y\, p\, X_{i+1} \ldots X_n$

In Case (1) of this definition we have added the condition $X_1 \ldots X_{i-2}X_{i-1} \neq \varepsilon$ because the tape head has to move to the left, and thus, 'before the move', it should *not* scan the leftmost cell of the tape.

When the transition function δ of a Turing Machine M is applied to the current state and the scanned symbol, we have that the current configuration γ_1 is changed into a new configuration γ_2. In this case we say that M *makes the move* (or *the computation step*, or *the step*) *from* γ_1 *to* γ_2 and we write $\gamma_1 \rightarrow_M \gamma_2$.

As usual, the reflexive and transitive closure of the relation $\rightarrow_M$ is denoted by $\rightarrow_M^*$.

The following definition introduces various concepts about the halting behaviour of a Turing Machine. They will be useful in the sequel.

Definition 7. [Final States and Halting Behaviour of a Turing Machine] (i) We say that a Turing Machine M *enters a final state* when making the move $\gamma_1 \rightarrow_M \gamma_2$ iff the state in the configuration γ_2 is a final state.

(ii) We say that a Turing Machine M *stops* (or *halts*) *in a configuration* $\alpha_1\, q\, \alpha_2$ iff no quintuple of M is of the form: $q,\, X \;\longmapsto\; q_j,\, X_k,\, m$, where X is the leftmost symbol of α_2, for some state $q_j \in Q$, symbol $X_k \in \mathit{\Gamma}$, and value $m \in \{L, R\}$. Thus, in this case no configuration γ exists such that $\alpha_1\, q\, \alpha_2 \rightarrow_M \gamma$.

(iii) We say that a Turing Machine M *stops* (or *halts*) *in a state* q iff no quintuple of M is of the form: $q,\, X_h \;\longmapsto\; q_j,\, X_k,\, m$ for some state $q_j \in Q$, symbols $X_h, X_k \in \mathit{\Gamma}$, and value $m \in \{L, R\}$.

(iv) We say that a Turing Machine M *stops* (or *halts*) *on the input* w iff for the initial configuration $q_0\, w$ there exists a configuration γ such that: (i) $q_0 w \rightarrow_M^* \gamma$, and (ii) M stops in the configuration γ.

(v) We say that a Turing Machine M *stops* (or *halts*) iff for every initial configuration $q_0\, w$ there exists a configuration γ such that: (i) $q_0 w \rightarrow_M^* \gamma$, and (ii) M stops in the configuration γ.

In Case (v), instead of saying: 'the Turing Machine M stops' (or halts), we also say: 'the Turing Machine M *always* stops' (or *always* halts, respectively). Indeed, we will do so when we want to stress the fact that M stops for all initial configurations of the form $q_0 w$, where q_0 is the initial state and w is an a input word.

Definition 8. [Language Accepted by a Turing Machine. Equivalence Between Turing Machines] Let us consider a Turing Machine M with initial state q_0, and an input word $w \in \Sigma^*$ for M.

(i) We say that M *answers 'yes' for* w (or M *accepts* w) iff (1.1) $q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, and $\alpha_2 \in \Gamma^+$, and (1.2) M stops in the configuration $\alpha_1\, q\, \alpha_2$.

(ii) We say that M *answers 'no' for* w (or M *rejects* w) iff (2.1) for all configurations γ such that $q_0 w \rightarrow_M^* \gamma$, the state in γ is *not a* final state, and (2.2) there exists a configuration γ such that $q_0 w \rightarrow_M^* \gamma$ and M stops in γ.

(iii) The set $\{w \mid w \in \Sigma^*$ and $q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, and $\alpha_2 \in \Gamma^+\}$ which is a subset of Σ^*, is said to be the *language accepted* by M and it denoted by $L(M)$. Every word in $L(M)$ is said to be a word *accepted* by M. A language accepted by a Turing Machine is said to be *Turing computable.*

(iv) Two Turing Machines M_1 and M_2 are said to be *equivalent* iff $L(M_1)$ and $L(M_2)$.

When the input word w is understood from the context, we will simply say: M answers 'yes' (or 'no'), instead of saying: M answers 'yes' (or 'no') for the word w.

Note that in other textbooks, when introducing the concepts of Definition 8 above, the authors use the expressions 'recognizes', 'recognized', and 'does not recognize', instead of 'accepts', 'accepted', and 'rejects', respectively.

Remark 1. **[Halting Hypothesis]** Unless otherwise specified, we will assume the following hypothesis, called the *Halting Hypothesis*:

for all Turing Machines M, for all initial configuration $q_0 w$, and for all configurations γ, *if* $q_0 w \rightarrow_M^* \gamma$ and the state in γ is final *then* no configuration γ' exists such that $\gamma \rightarrow_M \gamma'$.

Thus, by assuming the Halting Hypothesis, we will consider only Turing Machines which stop whenever they are in a final state. □

It is easy to see that this Halting Hypothesis can always be assumed without changing the notions introduced in the above Definition 8.

This means, in particular, that for every language L, we have that L is accepted by some Turing Machine M iff there exists a Turing Machine M_H such that for all words $w \in L$, (i) starting from the initial configuration $q_0 w$, *either* the state q_0 is a final state, *or* the state of the Turing Machine M_H will eventually be a final state, and (ii) whenever M_H is in a final state then M_H stops in that state.

For words which are *not* in L, the Turing Machine M_H may either (i) halt without ever being in a final state, or (ii) it may 'run forever', that is, may make an infinite number of moves without ever being in a final state.

As in the case of finite automata, we say that the notion of acceptance of a word w (or a language L) by a Turing Machine is *by final state*, because the word w (or every word of the language L, respectively) is accepted if the Turing Machine is in a final state or ever enters a final state.

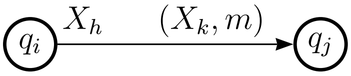
The notion of acceptance of a word, or a language, by a *nondeterministic* Turing Machine is identical to that of a deterministic Turing Machine. Thus, in particular, a word w is accepted by a nondeterministic Turing Machine M with initial state q_0, iff *there exists* a configuration γ such that $q_0 w \rightarrow_M^* \gamma$ and the state of γ is a final state.

Sometimes in the literature, one refers to this notion of acceptance by saying that every nondeterministic Turing Machine has *angelic nondeterminism.* The qualification 'angelic' is due to the fact that a word w is accepted by a nondeterministic Turing Machine M if *there exists* a sequence of moves (and not '*for all* sequences of moves') such that M goes from the initial configuration $q_0\, w$ to a configuration with a final state.

Analogously to what happens for finite automata, Turing Machines can be presented by giving their transition functions. Indeed, from the transition function of a Turing Machine M one can derive also its set of states, its input alphabet, and its tape alphabet. Moreover, the transition function δ of a Turing Machine can be represented as a graph, by representing each quintuple of δ of the form:

$$q_i,\, X_h \longmapsto q_j,\, X_k,\, m$$

as an arc from node q_i to a node q_j labelled by '$X_h \quad (X_k, m)$' as follows:



Remark 2. Without loss of generality, we may assume that the transition function δ of any given Turing Machine is a total function by adding a *sink state* to the set Q of states. We stipulate that: (i) the sink state is *not* final, and (ii) for every tape symbol which is in the scanned cell, the transition from the sink state takes the Turing Machine back to the sink state. □

Remark 3. We could allow the possibility of printing the blank symbol B. This possibility does not increase the power of a Turing Machine because the action of printing B, can be simulated by printing, instead, an extra symbol, say Δ, and then dealing with Δ as a given Turing Machine which can print B, deals with B. □

A Turing Machine M can be viewed as a device for computing a partial function from N^k to N in the sense of the following definition.

Definition 9. [Partial Functions from N^k to N Computed by Turing Machines] We say that a Turing Machine M with $\Sigma = \{0, 1\}$ computes a partial function f from N^k to N iff for every $n_1, \ldots, n_k \in N$ we have that:

(i) $q_0 0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0 \;\; \rightarrow^*_M \;\; \alpha_1\, q\, 0\, 1^{f(n_1,\ldots,n_k)}\, 0^t$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, $t \geq 1$, and

(ii) starting from the initial configuration $q_0 0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0$, the Turing Machine M goes through a sequence of configurations in which $\alpha_1\, q\, 0\, 1^{f(n_1,\ldots,n_k)}\, 0^t$ is the only configuration which includes a final state.

In this case we say that: (i) from the input sequence $0\, 1^{n_1}\, 0 \ldots 0\, 1^{n_k}\, 0$ the Turing Machine M computes the output sequence $0\, 1^{f(n_1,\ldots,n_k)}\, 0$, and (ii) the function f is *Turing computable*.

Remark 4. In the above Definition 9 we may also stipulate that if the Turing Machine M can print blanks, then t should be 1. Obviously, this extra condition does not modify the notion of a Turing computable function.

Example 1. The successor function $\lambda n.n{+}1$ is a Turing computable function. Indeed, the Turing Machine with the following transition function δ from the input sequence 01^n0 computes the output sequence $01^{n+1}0$:

$\delta(q_0,0) = (q_1,0,R)$;
$\delta(q_1,1) = (q_1,1,R)$; $\delta(q_1,0) = (q_1,1,R)$; $\delta(q_1,B) = (q_2,0,L)$;
$\delta(q_2,1) = (q_2,1,L)$; $\delta(q_2,0) = (q_3,0,R)$;
$\delta(q_3,1) = (q_F,1,L)$

where q_F is the only final state. This transition function δ is depicted in Figure 2. □

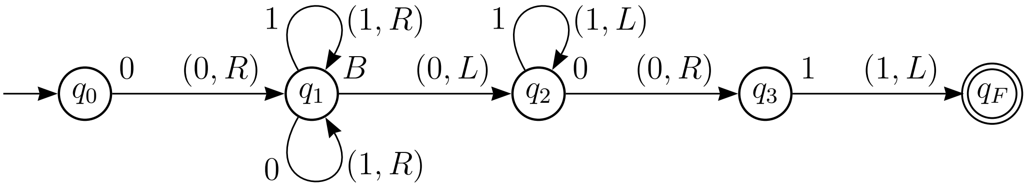


Fig. 2. A Turing Machine for computing the function $\lambda n.\, n{+}1$.

A Turing Machine M can also be viewed as a device for computing a partial function on strings in the sense indicated by the following definition.

Definition 10. [Partial Functions from Σ_1^* to Σ_2^* Computed by Turing Machines] Given the two alphabets Σ_1 and Σ_2 both subsets of $\Gamma - \{B\}$, we say that a Turing Machine M computes a partial function f from Σ_1^* to Σ_2^* iff for every $w \in \Sigma_1^*$ we have that:

(i) $q_0 w \;\rightarrow_M^*\; \alpha_1\, q\, b\, s$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, $b \in \Gamma$, and s in Σ_2^* with $s = f(w)$, and

(ii) for all γ if $q_0 w \;\rightarrow_M^*\; \gamma$ and $\gamma \neq \alpha_1\, q\, b\, s$, then the state in γ is *not* final (that is, starting from the initial configuration $q_0 w$, the Turing Machine M goes through a sequence of configurations in which $\alpha_1\, q\, b\, s$ is the only configuration with a final state).

The function f is said to be *Turing computable*.

If in this definition we stipulate that $q_0 w \;\rightarrow_M^*\; \alpha_1\, q\, s$, instead of $q_0 w \;\rightarrow_M^*\; \alpha_1\, q\, b\, s$, we get an equivalent definition. Definition 10 has the technical advantage of making some Turing Machine constructions a bit simpler (see the following Example 2).

In what follows, we will feel free to adopt different conventions for the encoding of the input and output strings.

We leave it to the reader to generalize Definition 10 so that a Turing Machine can be viewed as a devise for computing partial functions from $(\Sigma_1^*)^k$ to Σ_2^* for any $k \geq 0$. In particular, the k input strings can be written on the tape by using a separating, distinguished character not in $\Sigma_1 \cup \Sigma_2$.

Example 2. Let us consider the function f from $\{a\}^*$ to $\{0,1\}$ such that $f(w) = 0$ if the number of a's in w is even (that is, 0 or 2 or 4 or ...) and $f(w) = 1$ if the number of

a's in w is odd (that is, 1 or 3 or ...). This function is Turing computable by a Turing Machine M such that: $\Sigma = \{a\}$, $\Gamma = \{a, 0, 1, B\}$, and the transition function δ is given by the graph of Figure 3.

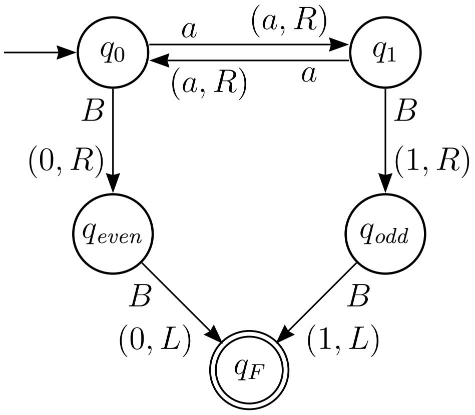


Fig. 3. A Turing Machine M for checking the length l of a string of a's. If l is even, M will eventually write 0, move left, and stop. If l is odd, M will eventually write 1, move left, and stop.

Let us see how the Turing Machine M of Example 2 works if its input w is the empty string ε. Here is the corresponding sequence of tape configurations where ▲ denotes the position of the tape head:

1:

B	B	B	B	...
▲				

2:

0	B	B	B	...
	▲			

3:

0	0	B	B	...
▲				

In configuration 3 the 0 in the cell to the right of the scanned cell shows that the number of a's in w is even. Here is the sequence of tape configurations if the input w of the Turing Machine M of Example 2 is aaa:

1:

a	a	a	B	B	...
▲					

2:

a	a	a	B	B	...
	▲				

3:

a	a	a	B	B	...
		▲			

4:

a	a	a	B	B	...
			▲		

5:

a	a	a	1	B	...
				▲	

6:

a	a	a	1	1	...
			▲		

In configuration 6 the 1 in the cell to the right of the scanned cell shows that the number of a's in w is odd.

We leave it as an exercise to the reader to show that in the Turing Machine of Figure 3, instead of 'B $(0, R)$', the label of the transition from state q_0 to state q_{even} can be any one of the form 'B (x, R)' for any $x \in \Gamma$. □

In what follows we will introduce the set of the partial recursive functions and in Section 17 we will state two theorems which identify the set of the partial recursive functions with the set of the Turing computable functions, both in the case of functions from N^k to N and in the case of functions from $(\Sigma_1^*)^k$ to Σ_2^*, for any $k \geq 0$.

Remark 5. We could have allowed three moves of the tape head: (i) L (*left*), (ii) R (*right*), and (iii) S (*stationary*), instead of the moves L and R only. This does not determines an increase of the power of the Turing Machine because the concatenation of some consecutive *stationary* moves followed by a *non-stationary* move, is equivalent to a *left* or a *right* move. Indeed, in a *stationary* move, after printing the new symbol on the tape, we know the new state of the Turing Machine and the new symbol which is read (which is the one that the Turing Machine has just printed) and, thus, we also know the configuration of the Turing Machine after its next move. □

Remark 6. In the literature [8] there is also a formalization of the Turing Machine via 4-tuples, instead of 5-tuples, in the sense that at each move of the Turing Machine: (i) either it writes of a symbol, (ii) or it moves one square to the left, (iii) or it moves one square to the right. In this formalization the Turing Machine can print the blank symbol. Any 4-tuple consists of: (i) the old state, (ii) the read symbol, (iii) the printed symbol or the move (to the left or to the right), and (iv) the new state. □

3 Techniques for Constructing Turing Machines

In this section we will present some techniques for constructing Turing Machines. The basic ideas will be presented through various examples.

3.1 Use of States to Store Information

The states of a Turing Machine can be used for storing information. We will illustrate this technique by looking at the following example. Let us assume that we want to construct a Turing Machine to check whether or not an input string belongs to the language $\{01^n \mid n \geq 0\} \cup \{10^n \mid n \geq 0\}$. If this is the case then the Turing Machine should enter a final state and an 1 should be in the cell to the right of the scanned cell.

Initially, in the tape we have the given string in $\{0,1\}^*$, and the cells $c_1\, c_2 \ldots c_{n+1}$ hold a string of length $n{+}1$ for some $n \geq 0$. If that string is of the form 01^n or 10^n for some $n \geq 0$ then and only then the check is positive.

Here are the sets defining the desired Turing Machine.

$Q = \{\langle p_0, a\rangle, \langle p_1, 0\rangle, \langle p_1, 1\rangle, \langle p_1, a\rangle\} \subseteq \{p_0, p_1\} \times \{0, 1, a\}$,
$\Sigma = \{0, 1\}$,
$\Gamma = \{0, 1, B\}$,
$q_0 = \langle p_0, a\rangle$, and
$F = \{\langle p_1, a\rangle\}$.

The transition function δ is defined as depicted in Figure 4. The second component of the state is for storing the information about the first symbol of the input string which is either 0 or 1.

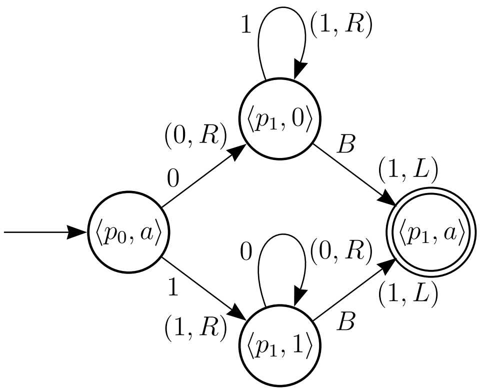


Fig. 4. A Turing Machine for testing whether or not a given string is an element of $\{01^n \mid n \geq 0\} \cup \{10^n \mid n \geq 0\}$.

3.2 Use of Multiple Tracks

If we have more than one track on the tape of a Turing Machine, it is sometimes easier to construct a Turing Machine which solves a given problem. Thus, it is useful in practice to consider Turing Machines with tapes which have multiple tracks. Note, however, that the assumption that the tape has k tracks, for $k \geq 1$, is equivalent to the assumption that we have new symbols which are k-tuples of old symbols. The tape head looks at a k-tuple at a time. The new blank symbol is the tuple of k old blank symbols.

To have multiple tracks is useful when one has to perform various destructive operations on the same input string as shown by the following example.

Let us assume that we want to construct a Turing Machine for testing whether or not a given number is divisible by 3 and 5 (that is, by 15). We may consider three tracks, like, for instance, the following ones:

track 1: $\$\ 1\ 0\ 1\ 0\ 1\ 1\ 1\,0\ \$\ B\ B \ldots$

track 2: $B\ 1\ 0\ 1\ 0\ 1\ 0\ 1\,1\ B\ B\ B \ldots$

track 3: $B\ B\ B\ B\ B\ B\ B\ 1\,1\ B\ B\ B \ldots$

The given number is written in binary on the first track between \$ symbols. The input symbols are: $\begin{vmatrix}\$\\B\\B\end{vmatrix}$, $\begin{vmatrix}0\\B\\B\end{vmatrix}$, and $\begin{vmatrix}1\\B\\B\end{vmatrix}$ (in the three-track tape above, each symbol is represented in a vertical way by its three components).

The tape symbols are like the input symbols, but they may have either 0 or 1 or B in the second and in the third track.

One can test the divisibility by 3 and 5 of the given number: (i) by writing 3 in binary on the third track, (ii) by copying the input number on the second track, and (iii) repeatedly subtract 3 from the second track until 0 is found. Then, one should repeat this process by writing 5 in binary on the third track and copying the input number (which is kept unchanged on the first track) on the second track, and repeatedly subtract 5 from the second track until 0 is found. In this case and only in this case, the given number is divisible by 3 and 5.

For instance, copying the input on the second track is performed by quintuples of the form:

$$q,\ \begin{vmatrix} 1 \\ y \\ z \end{vmatrix} \;\rightarrow\; q,\ \begin{vmatrix} 1 \\ 1 \\ z \end{vmatrix},\ R \quad \text{for any } y, z \in \{B, 0, 1\}$$

$$q,\ \begin{vmatrix} 0 \\ y \\ z \end{vmatrix} \;\rightarrow\; q,\ \begin{vmatrix} 0 \\ 0 \\ z \end{vmatrix},\ R \quad \text{for any } y, z \in \{B, 0, 1\}$$

We leave it to the reader as an exercise to complete the description of the desired Turing Machine.

3.3 Test of String Equality

In this section we present an example where we construct a Turing Machine which computes the function f from $\{0,1\}^* \times \{0,1\}^*$ to $\{0,1\}$ such that

$$f(x,y) = \text{if } x\!=\!y \text{ then } 1 \text{ else } 0. \qquad (\dagger)$$

This Turing Machine has a tape with two tracks and thus, we assume that the symbols of the input alphabet and the tape alphabet are pairs. Here is the definition of the desired Turing Machine M.

$Q = \{p_1, p_{10}, p_{11}, p_{20}, p_{21}, p_3, p_4, p_5, p_6\}$,

$\Sigma = \left\{ \begin{vmatrix} 0 \\ B \end{vmatrix}, \begin{vmatrix} 1 \\ B \end{vmatrix}, \begin{vmatrix} \# \\ B \end{vmatrix} \right\}$,

$\Gamma = \left\{ \begin{vmatrix} 0 \\ B \end{vmatrix}, \begin{vmatrix} 0 \\ \checkmark \end{vmatrix}, \begin{vmatrix} 1 \\ B \end{vmatrix}, \begin{vmatrix} 1 \\ \checkmark \end{vmatrix}, \begin{vmatrix} \# \\ B \end{vmatrix}, \begin{vmatrix} B \\ B \end{vmatrix} \right\}$,

the initial state is p_1,

the blank symbol is $\begin{vmatrix} B \\ B \end{vmatrix}$, and

$F = \{p_6\}$.

The symbols 0 and 1 of the specification (†) are represented within the Turing Machine M by $\begin{vmatrix} 0 \\ B \end{vmatrix}$ and $\begin{vmatrix} 1 \\ B \end{vmatrix}$, respectively. As usual, the output symbol will be written to the right of the cell scanned by the tape head when the final state is entered. The input strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$ are represented on the leftmost position of the tape as follows:

track 1: $\quad x_1\; x_2 \ldots x_n\; \#\; y_1\; y_2 \ldots y_m\; B\; B \ldots$
track 2: $\quad B\; B\; B\;\; B\; B\; B\; B\;\; B\;\; B\; B\; B \ldots$

and initially, the tape head looks at the leftmost cell where $\begin{vmatrix} x_1 \\ B \end{vmatrix}$ is placed.

The transition function δ is defined as follows. First of all, the Turing Machine M checks whether or not $n\!=\!0$. If $n\!=\!0$ then M checks that also $m\!=\!0$ and, if this is the case, it returns $\left|\begin{array}{c}1\\B\end{array}\right|$ to the right of the cell scanned by the head when the final state is entered (see quintuples 1 and 2 below).

If $n\!>\!0$ and $x_1\!=\!0$ (the case when $x_1\!=\!1$ is analogous) then M checks it off by writing $\left|\begin{array}{c}0\\\checkmark\end{array}\right|$, moves to the right, passes $\left|\begin{array}{c}\#\\B\end{array}\right|$, and looks for the leftmost unchecked symbol of y which should be 0. If M finds it, M checks it off by writing $\left|\begin{array}{c}0\\\checkmark\end{array}\right|$ and moves to the left, passing over $\left|\begin{array}{c}\#\\B\end{array}\right|$ and looks for the leftmost unchecked symbol of x.

Then the Turing Machine M starts the process again until the leftmost unchecked symbol is $\left|\begin{array}{c}\#\\B\end{array}\right|$. At that point it makes sure that all symbols to the right of $\left|\begin{array}{c}\#\\B\end{array}\right|$ are checked (that is, they have on the lower track $\checkmark$) and it terminates successfully (that is, it writes $\left|\begin{array}{c}1\\B\end{array}\right|$, moves to the left, and stops). In all other cases the termination of the Turing Machine is unsuccessful (that is, it writes $\left|\begin{array}{c}0\\B\end{array}\right|$, moves to the left, and stop).

We have the following quintuples:

1. $p_1, \left|\begin{array}{c}\#\\B\end{array}\right| \rightarrow p_5, \left|\begin{array}{c}\#\\B\end{array}\right|, R$ 2. $p_5, \left|\begin{array}{c}B\\B\end{array}\right| \rightarrow p_6, \left|\begin{array}{c}1\\B\end{array}\right|, L$

The following two quintuples look for $\left|\begin{array}{c}B\\B\end{array}\right|$ immediately to the right of the checked symbols:

3. $p_5, \left|\begin{array}{c}0\\\checkmark\end{array}\right| \rightarrow p_5, \left|\begin{array}{c}0\\\checkmark\end{array}\right|, R$ 4. $p_5, \left|\begin{array}{c}1\\\checkmark\end{array}\right| \rightarrow p_5, \left|\begin{array}{c}1\\\checkmark\end{array}\right|, R$

Note that in state p_5 if some symbols of y are not checked then $x \neq y$ because, if this the case, the strings x and y have different length.

The following quintuples are needed for checking off symbols on the strings x and y. If the symbol of x to be checked is 0 we use the quintuples 0.1–0.5, if it is 1 we use the quintuples 1.1–1.5.

0.1 $p_1, \left|\begin{array}{c}0\\B\end{array}\right| \rightarrow p_{10}, \left|\begin{array}{c}0\\\checkmark\end{array}\right|, R$ Checking off the leftmost unchecked symbol of x.
Let it be 0.

0.2 $p_{10}, \left|\begin{array}{c}s\\B\end{array}\right| \rightarrow p_{10}, \left|\begin{array}{c}s\\B\end{array}\right|, R$ Moving over to y. (2 quintuples: for $s = 0, 1$)

0.3 $p_{10}, \left|\begin{array}{c}\#\\B\end{array}\right| \rightarrow p_{20}, \left|\begin{array}{c}\#\\B\end{array}\right|, R$ Passing # and going over to y.

0.4 $p_{20}, \left|\begin{array}{c}s\\\checkmark\end{array}\right| \rightarrow p_{20}, \left|\begin{array}{c}s\\\checkmark\end{array}\right|, R$ Looking for the leftmost unchecked symbol of y.
(2 quintuples: for $s = 0, 1$)

0.5 $p_{20}, \left|\begin{array}{c}0\\B\end{array}\right| \rightarrow p_3, \left|\begin{array}{c}0\\\checkmark\end{array}\right|, L$ Now we go back to string x.

1.1 $p_1, \left|\begin{matrix}1\\B\end{matrix}\right| \rightarrow p_{11}, \left|\begin{matrix}1\\\checkmark\end{matrix}\right|, R$ Checking off the leftmost unchecked symbol of x.
Let it be 1.

1.2 $p_{11}, \left|\begin{matrix}s\\B\end{matrix}\right| \rightarrow p_{11}, \left|\begin{matrix}s\\B\end{matrix}\right|, R$ Moving over to y. (2 quintuples: for $s=0,1$)

1.3 $p_{11}, \left|\begin{matrix}\#\\B\end{matrix}\right| \rightarrow p_{21}, \left|\begin{matrix}\#\\B\end{matrix}\right|, R$ Passing # and going over to y.

1.4 $p_{21}, \left|\begin{matrix}s\\\checkmark\end{matrix}\right| \rightarrow p_{21}, \left|\begin{matrix}s\\\checkmark\end{matrix}\right|, R$ Looking for the leftmost unchecked symbol of y.
(2 quintuples: for $s=0,1$)

1.5 $p_{21}, \left|\begin{matrix}1\\B\end{matrix}\right| \rightarrow p_{3}, \left|\begin{matrix}1\\\checkmark\end{matrix}\right|, L$ Now we go back to string x.

The following quintuples allow us to go back to the string x and to reach the leftmost unchecked symbol of x, if any. Then we check off that symbol of x.

5. $p_3, \left|\begin{matrix}s\\\checkmark\end{matrix}\right| \rightarrow p_{3}, \left|\begin{matrix}s\\\checkmark\end{matrix}\right|, L$ Moving over to x. (2 quintuples: for $s=0,1$)

6. $p_3, \left|\begin{matrix}\#\\B\end{matrix}\right| \rightarrow p_{4}, \left|\begin{matrix}\#\\B\end{matrix}\right|, L$ Passing # and going over to x.

7. $p_4, \left|\begin{matrix}s\\B\end{matrix}\right| \rightarrow p_{4}, \left|\begin{matrix}s\\B\end{matrix}\right|, L$ Looking for the leftmost unchecked symbol of x.
(2 quintuples: for $s=0,1$)

8. $p_4, \left|\begin{matrix}s\\\checkmark\end{matrix}\right| \rightarrow p_{1}, \left|\begin{matrix}s\\\checkmark\end{matrix}\right|, R$ Ready to check off one more symbol of x.
(2 quintuples: for $s=0,1$)

All undefined transitions should be made to go to the final state p_6 and the symbol to the right of the scanned cell should be $\left|\begin{matrix}0\\B\end{matrix}\right|$, denoting that $x \neq y$.

For instance, for $x=y=0010$, we have the following sequence of tape configurations with two tracks. The initial configuration is:

$$\begin{array}{l}0\ \ 0\ \ 1\ \ 0\ \#\ 0\ \ 0\ \ 1\ \ 0\ B\,B\ldots\\ B\,B\,B\,B\,B\,B\,B\,B\,B\,B\,B\ldots\end{array}$$

then we get:

$$\begin{array}{l}0\ \ 0\ \ 1\ \ 0\ \#\ 0\ \ 0\ \ 1\ \ 0\ B\,B\ldots\\ \checkmark\ B\,B\,B\,B\,B\,B\,B\,B\,B\,B\ldots\end{array}$$

After some moves to the right, having checked the equality of the first symbols of the two sequences, we have:

$$\begin{array}{l}0\ \ 0\ \ 1\ \ 0\ \#\ 0\ \ 0\ \ 1\ \ 0\ B\,B\ldots\\ \checkmark\ B\,B\,B\,B\ \checkmark\ B\,B\,B\,B\,B\ldots\end{array}$$

and after some moves to the left, looking for the second symbols to check, we have:

$$\begin{array}{l}0\ \ 0\ \ 1\ \ 0\ \#\ 0\ \ 0\ \ 1\ \ 0\ B\,B\ldots\\ \checkmark\ \checkmark\ B\,B\,B\ \checkmark\ B\,B\,B\,B\,B\ldots\end{array}$$

and so on, until at the end, we have:

$$\begin{array}{cccccccccccc} 0 & 0 & 1 & 0 & \# & 0 & 0 & 1 & 0 & 1 & B & \ldots \\ \checkmark & \checkmark & \checkmark & \checkmark & B & \checkmark & \checkmark & \checkmark & \underset{\blacktriangle}{\checkmark} & B & B & \ldots \end{array}$$

and the tape head is at the cell marked with ▲.

3.4 Shift of Strings

A Turing Machine may shift the content of its tape of $k\,(>0)$ cells to the right (or to the left, if there are cells available). Let us consider the case of a right shift of 1 cell. This shift can be obtained by: (i) going to the leftmost cell c_1, (ii) storing in a state of the control the symbol in c_1, (iii) printing in c_1 a symbol to denote that the cell is empty, and while moving to the right, (iv) storing in a state of the control the symbol read in the scanned cell, and (v) printing in the scanned cell the symbol which was to the left of it and was stored in a state of the control during the previous move.

When a Turing Machine shifts the content of its tape, we say that it has applied the *shifting-over* technique.

3.5 Use of Subroutines

We restrict our attention to subroutines without recursion and without parameters. Later on in Section 8 we shall see how to overcome this limitation by explaining in an informal way how a Turing Machine can simulate a Random Access Machine. If a Turing Machine, say S, is a subroutine for another Turing Machine, say M, we first require that the set of states of S is disjoint from the set of states of M. When a call to the subroutine S is made, the initial state of S is entered and when the subroutine S has finished, M enters one of its states and proceeds.

We present the idea of using Turing Machines as subroutines through the following example. Let Σ be $\{0,1\}$. Let us consider the Turing Machine *Sum* which takes the initial tape configuration:

$$\ldots \underset{\blacktriangle}{\$}\ \lceil n \rceil\ \#\ \lceil m \rceil\ \$ \ldots$$

where $\lceil n \rceil$ and $\lceil m \rceil$ are strings of bits (each bit in a separate cell) denoting the two non-negative numbers n and m written in binary (using enough bits to allow the process to continue as specified below). The machine *Sum* derives the final tape configuration:

$$\ldots \$\ \varDelta\ \varDelta \ldots \varDelta\ \lceil n{+}m \rceil\ \underset{\blacktriangle}{\$} \ldots$$

where $\varDelta$'s are printable blanks. We also assume that *Sum* never goes outside the tape section between \$'s. The Turing Machine *Mult* for multiplying two natural numbers can be constructed by making use of the subroutine *Sum* as follows.

(1) *Mult* starts from the tape configuration:

$$\ldots \lceil n \rceil\ \#\ \lceil m \rceil\ \underset{\blacktriangle}{\$}\ \lceil 0 \rceil\ \#\ \lceil 0 \rceil\ \$ \ldots$$

(2) *Mult* gives control to *Sum* and eventually we get:

$$\ldots \lceil n \rceil\ \#\ \lceil m \rceil\ \$\ \varDelta\ \varDelta \ldots \varDelta\ \lceil s \rceil\ \underset{\blacktriangle}{\$} \ldots$$

where s is the sum of the numbers between \$'s. Initially, we have that s is equal to 0.

```
(3) Sum returns the control to Mult, and
if $n=0$ then Mult produces the result $s$
else begin Mult derives $\lceil n-1\rceil$ from $\lceil n\rceil$ and copies $\lceil m\rceil$
            so that eventually we get the tape configuration:
            $\ldots \lceil n-1\rceil \; \# \; \lceil m\rceil \; \$ \; \lceil m\rceil \; \# \; \lceil s\rceil \; \$ \ldots$
                                  ▲
            and then Mult goes back to (2)
     end
```

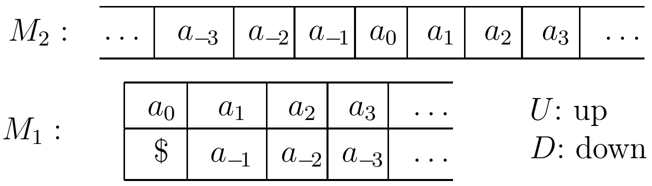
We leave it as an exercise to the reader to complete the details of the construction of the Turing Machine *Mult*.

4 Extensions of Turing Machines Which Do Not Increase Their Power

4.1 Two-way Infinite Tape

We may assume a different definition of Turing Machine where the tape is infinite in both directions. We have that a Turing Machine M_2 whose tape is infinite in both directions, can be simulated by a Turing Machine M_1 whose tape is one-way infinite as implicitly specified by Definition 3 on page 11 and Definition 8 on page 14.

Let $M_2 = \langle Q_2, \Sigma_2, \Gamma_2, \delta_2, q_2, B, F_2\rangle$ be the given Turing Machine with the two-way infinite tape. The following Turing Machine $M_1 = \langle Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, B, F_1\rangle$ with the one-way infinite tape with two tracks simulates M_2 as we now indicate (see also Figure 5).



M_2 :

...	a_{-3}	a_{-2}	a_{-1}	a_0	a_1	a_2	a_3	...

M_1 :

a_0	a_1	a_2	a_3	...
\$	a_{-1}	a_{-2}	a_{-3}	...

U: up
D: down

Fig. 5. Simulating a two-way infinite tape with two tracks by a one-way infinite tape.

For the Turing Machine M_1 we stipulate that:

$Q_1 = \{\langle q, U\rangle, \langle q, D\rangle \mid q \in Q_2\} \,\cup\, \{q_1\}$,

$\Sigma_1 = \{\left|\begin{smallmatrix} a \\ B \end{smallmatrix}\right| \mid a \in \Sigma_2\}$,

$\Gamma_1 = \{\left|\begin{smallmatrix} a \\ b \end{smallmatrix}\right| \mid a, b \in \Gamma_2\} \,\cup\, \{\left|\begin{smallmatrix} a \\ \$ \end{smallmatrix}\right| \mid a \in \Gamma_2\}$ where \$ is not in Γ_2,

the initial state is q_1, and

$F_1 = \{\langle q, U\rangle, \langle q, D\rangle \mid q \in F_2\}$.

U stands for ‘up’ and D for ‘down’, corresponding to the upper and lower track, respectively.

Now, we assume that if the input string of the machine M_2 is placed on the cells, say $0, 1, \ldots, n$ with symbols $a_0, a_1, \ldots, a_n$, respectively (see Figure 5) and all remaining cells have blanks, then in the machine M_1 the input string is placed in leftmost cells with content $\left|\begin{matrix} a_0 \\ B \end{matrix}\right|, \left|\begin{matrix} a_1 \\ B \end{matrix}\right|, \ldots, \left|\begin{matrix} a_n \\ B \end{matrix}\right|$, respectively. We also assume that the tape head of M_2 is at the cell 0 and that of M_1 is at its leftmost cell, as usual. The moves of the machine M_1 are defined in terms of the moves of M_2 according to the following rules.

1. Initial moves of M_1:

for each $a \in \Sigma_2 \cup \{B\},\ \ X \in \Gamma_2 - \{B\},\ \ q \in Q_2,$

$q_1, \left|\begin{matrix} a \\ B \end{matrix}\right| \;\rightarrow\; \langle q, U\rangle, \left|\begin{matrix} X \\ \$ \end{matrix}\right|, R$ if in M_2 we have: $q_2, a \;\rightarrow\; q, X, R$

$q_1, \left|\begin{matrix} a \\ B \end{matrix}\right| \;\rightarrow\; \langle q, D\rangle, \left|\begin{matrix} X \\ \$ \end{matrix}\right|, R$ if in M_2 we have: $q_2, a \;\rightarrow\; q, X, L$

M_1 prints $\$$ on the leftmost cell of the lower track and continues working as M_2.

2.1 M_1 simulates the moves of M_2 on the upper track:

for each $\left|\begin{matrix} a \\ Y \end{matrix}\right| \in \Gamma_1$ with $Y \neq \$,\ \ b \in \Gamma_2 - \{B\},\ \ m \in \{L, R\},\ \ q, p \in Q_2,$

$\langle q, U\rangle, \left|\begin{matrix} a \\ Y \end{matrix}\right| \;\rightarrow\; \langle p, U\rangle, \left|\begin{matrix} b \\ Y \end{matrix}\right|, m$ if in M_2 we have: $q, a \;\rightarrow\; p, b, m$

2.2 M_1 simulates the moves of M_2 on the lower track:

for each $\left|\begin{matrix} Y \\ a \end{matrix}\right| \in \Gamma_1,\ \ b \in \Gamma_2 - \{B\},\ \ q, p \in Q_2,$

$\langle q, D\rangle, \left|\begin{matrix} Y \\ a \end{matrix}\right| \;\rightarrow\; \langle p, D\rangle, \left|\begin{matrix} Y \\ b \end{matrix}\right|, R$ if in M_2 we have: $q, a \;\rightarrow\; p, b, L$

$\langle q, D\rangle, \left|\begin{matrix} Y \\ a \end{matrix}\right| \;\rightarrow\; \langle p, D\rangle, \left|\begin{matrix} Y \\ b \end{matrix}\right|, L$ if in M_2 we have: $q, a \;\rightarrow\; p, b, R$

3. Possible change of track:

for each $a, b \in \Gamma_2 - \{B\},\ \ q, p \in Q_2,$

$\langle q, U\rangle, \left|\begin{matrix} a \\ \$ \end{matrix}\right| \;\rightarrow\; \langle p, D\rangle, \left|\begin{matrix} b \\ \$ \end{matrix}\right|, R$ if in M_2 we have: $q, a \;\rightarrow\; p, b, L$
(from the upper track to the lower track)

$\langle q, U\rangle, \left|\begin{matrix} a \\ \$ \end{matrix}\right| \;\rightarrow\; \langle p, U\rangle, \left|\begin{matrix} b \\ \$ \end{matrix}\right|, R$ if in M_2 we have: $q, a \;\rightarrow\; p, b, R$
(remaining in the upper track)

$\langle q, D\rangle, \left|\begin{matrix} a \\ \$ \end{matrix}\right| \;\rightarrow\; \langle p, D\rangle, \left|\begin{matrix} b \\ \$ \end{matrix}\right|, R$ if in M_2 we have: $q, a \;\rightarrow\; p, b, L$
(remaining in the lower track)

$\langle q, D\rangle, \left|\begin{matrix} a \\ \$ \end{matrix}\right| \;\rightarrow\; \langle p, U\rangle, \left|\begin{matrix} b \\ \$ \end{matrix}\right|, R$ if in M_2 we have: $q, a \;\rightarrow\; p, b, R$
(from the lower track to the upper track)

We leave it to the reader to convince himself that M_1 is equivalent to M_2 (see Definition 8 on page 14).

4.2 Multitape Turing Machines

A *multitape* Turing Machine has k tapes with $k > 1$, and each tape has its own head. A move depends on the k-tuple of symbols read by the heads. In a single move k new symbols are printed (they may be all different), one on each tape, and the heads are moved, left or right, independently. We assume that the input is written on a given tape, say tape 1, and the tapes 2, 3, ..., and k are all filled with blanks.

Let us informally explain how a multitape Turing Machine can be simulated by a single tape Turing Machine.

A k-tape Turing Machine M_k is simulated by a $2k$-track Turing Machine M, each tape being represented by two tracks: on the upper track there is a copy of the corresponding tape and on the lower track there is either a blank symbol or the 'head marker' ▲ to denote the position of the tape head (see Figure 6 where we did not show the blank symbols). Initially the tape head of M is on the position corresponding to the leftmost marker ▲. The internal states of M hold the information of both (i) the internal states of M_k and (ii) the count of the head markers to the left of the head of M itself.

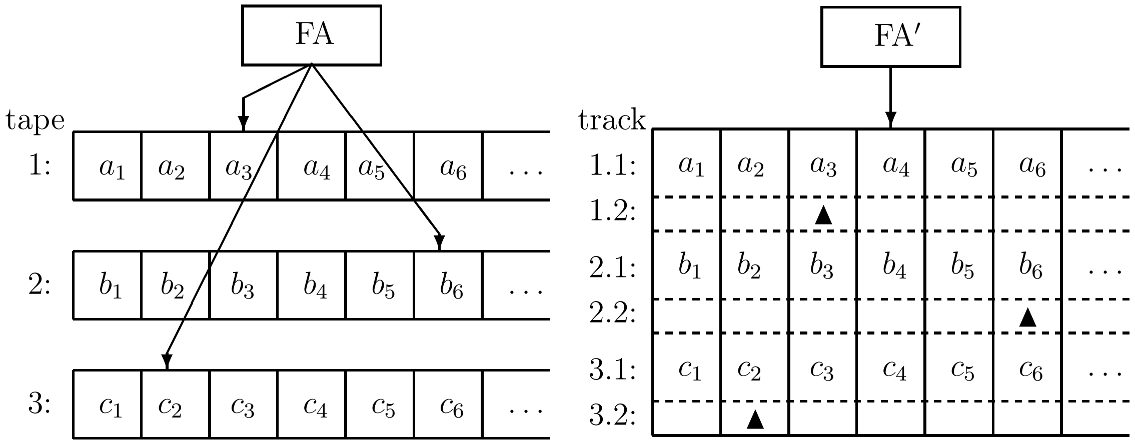

FA
tape
1:
a_1 a_2 a_3 a_4 a_5 a_6 ...
2:
b_1 b_2 b_3 b_4 b_5 b_6 ...
3:
c_1 c_2 c_3 c_4 c_5 c_6 ...
FA′
track
1.1:
a_1 a_2 a_3 a_4 a_5 a_6 ...
1.2:
▲
2.1:
b_1 b_2 b_3 b_4 b_5 b_6 ...
2.2:
▲
3.1:
c_1 c_2 c_3 c_4 c_5 c_6 ...
3.2:
▲

Fig. 6. Simulating a 3-tape Turing Machine by a 6-track Turing Machine.

A move m of M_k is simulated by the Turing Machine M as follows. M moves its head from the leftmost to the rightmost head marker and then back to the leftmost head marker. While the move to the right proceeds, M recalls in its internal states the number of head markers passed, the symbols scanned by each head marker. When all head markers have been passed, M has the necessary information for simulating the move m of M_k. Thus, M changes its internal state according to the move m to be simulated, and makes a leftwards movement until the leftmost head marker is reached. During that movement M updates its tape configuration (that is, updating both the symbols scanned by the head markers and the positions of the head markers) according to the move m. As done in the movement to the right, in its movement to the left M counts in its internal states the number of the head markers visited and then M will know when to stop its movement to the left. Having simulated the printing and the head movements, at the end M changes its internal state to simulate the change of the internal state of M_k.

It should be noticed that when moving back to the left, M may be forced to make some one-cell movements to the right for simulating one-cell movements to the right of some of the heads of M_k.

In the simulation we have described, M requires many moves for simulating a single move of M_k. Indeed, in this simulation (unlike the simulation of the two-way infinite tape Turing Machine by the one-way infinite tape Turing Machine) there is a slowdown, because at each move which is simulated the distance of the leftmost head from the rightmost head may increase by 2 cells. Thus, the head of the simulating machine M may have to travel $2r$ cells when simulating the r-th move of M_k.

The slowdown is quadratic because: (i) if we disregard the distance to travel, the number of transitions to be made by M for simulating the moves of M_k is constant, being k fixed, and (ii) the sum of the distances after r moves, apart from some constants which we do not take into consideration, is $2+4+6+\ldots+2r$, that is, $O(r^2)$.

4.3 Multihead Turing Machines

Multihead Turing Machines have many heads which move independently of each other, and the transition of the machine depends on the internal state and on the symbols which are scanned. A quintuple of a Turing Machine with r heads looks as follows:

$$q, \langle x_1, \ldots x_r \rangle \;\rightarrow\; p, \langle y_1, \ldots y_r \rangle, \langle m_1, \ldots m_r \rangle$$

where for $i = 1, \ldots, r$, x_i and y_i are tape symbols, and m_i is an element of $\{L, R\}$.

The simulation of this kind of Turing Machines by a Turing Machine with one head only is similar to the one of the multitape Turing Machine. The only difference is that now the simulating one-head Turing Machine need $k+1$ tracks, k tracks for the k heads and one track for the tape.

4.4 Off-line and On-line Turing Machines

In this section we introduce two more kinds of Turing Machines which are considered in the literature.

The first one, called the *off-line Turing Machine*, is considered when studying Complexity Theory and, in particular, when analyzing the amount of space required by Turing Machine computations.

An off-line Turing Machine is a two-tape Turing Machine with the limitation that one of the two tapes, called the *input tape*, is a tape which contains the input word between two special symbols, say ¢ and \$. The input tape can be read, but not modified. Moreover, it is not allowed to use the input tape outside the cells where the input is written (see Figure 7). The other tape of an off-line Turing Machine will be referred to as the *standard tape* (or the *working tape*).

An off-line Turing Machine is equivalent to a standard Turing Machine (that is, the one of Definition 3) because: (i) an off-line Turing Machine is a special case of a two-tape Turing Machine, which is equivalent to a standard Turing Machine, and (ii) for any given Turing Machine M and input w there exists an off-line Turing Machine M_{off} with input

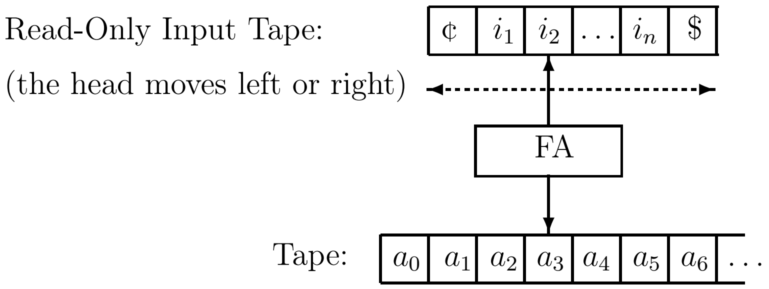


Fig. 7. An off-line Turing (the lower tape may equivalently be two-way infinite or one-way infinite).

w which can simulate M on the input w. Indeed, M_{off} first copies its input which is on its input tape, onto the other tape, and then M_{off} behaves as M does.

The second kind of Turing Machine we consider in this section is the *on-line Turing Machine*. This kind of Turing Machine is used for studying the so called *on-line algorithms* [14] and their complexity.

The on-line Turing Machine is like the off-line Turing Machine with the assumption that the head on the input tape initially scans the symbol ¢ and then it can remain stationary or move to the right. It *cannot* move to the left. When the head on the input tape moves to the right and by that move it scans $, then the finite control of the Turing Machine enters a state which is *either* a final state (and this is the case when the on-line Turing Machine accepts its input) *or* a non-final state (and this is the case when the on-line Turing Machine does not accept its input). Other equivalent conventions on the final reading move are described in the literature.

4.5 Bidimensional and Multidimensional Turing Machines

In this section we look at a Turing Machine with a bidimensional tape. The cases with multidimensional tapes are similar. The head of the machine may move left, right, up, and down, and before moving it prints a symbol on the scanned cell. At each moment there exists a rectangle outside which all cells are filled with blanks. (see Figure 8)

B	r	e	s	B	B	B
d	s	a	u	B	k	B
B	B	t	n	B	B	B
B	B	B	h	B	B	B

Fig. 8. The non-blank portion of the tape of a two-dimensional Turing Machine.

A bidimensional Turing Machine M_2 can be simulated by a standard Turing Machine M_1 with two tapes as follows. The need for the second tape is simply for explanatory reasons, and it related to the fact that for the simulation we need to count the number of the special symbols $*$ and \$ (see below). The basic idea is to simulate *line-by-line*, *top-down* a bidimensional configuration by a linear configuration. For instance, a possible line-by-line, top-down encoding of the tape configuration of Figure 8 is:

$$\$\, B\, r\, e\, s\, B\, B\, B * d\, s\, a\, u\, B\, k\, B * B\, B\, t\, n\, B\, B\, B * B\, B\, B\, h\, B\, B\, B\, \$$$

where the symbol $*$ is a new symbol, not in the tape alphabet of M_2. This line-by-line representation is the usual one for television images. $*$ denotes the end of a line. \$ denotes, so to speak, the upper left corner and the lower right corner of the rectangle. We may assume that the input to the machine M_2 is given a sequence of symbols in one dimension only, say the horizontal dimension. Horizontal moves of M_2 inside the current rectangle are simulated in the obvious way by the Turing Machine M_1. Vertical moves inside the current rectangle can be simulated by counting the distance from the nearest left $*$ so that it is possible to place the head on the proper position in the upper or lower line.

If a move goes outside the current rectangle and it is vertical and upwards then M_1 replaces \$ to the left of the current configuration by $\$\, B\, B\, \ldots\, B\, *$, where the number of B's are counted in the second tape by finding first the length of a line. This operation corresponds to the insertion of a new line.

The case in which the move is downwards is similar, but the machine M_1 should operate at the right end of the configuration.

If the move of M_2 which goes outside the rectangle is to the right then M_1 should first add an extra blank at the end of each line (and this can be done by shifting) and then it simulates the move of M_2. Analogously for the case when the move is to the left.

This simulation of the bidimensional Turing Machine can be extended to the case when there are many heads or many multidimensional tapes by combining together the various techniques presented in the previous sections.

4.6 Nondeterministic versus Deterministic Turing Machines

Now we show that nondeterministic Turing Machines are equivalent to deterministic Turing Machines (that is, the ones of Definition 3).

A nondeterministic Turing Machine M is like a standard Turing Machine (that is the one of Definition 3), but the condition that the transition function δ be a partial function is released in the sense that there can be more that one quintuple with the same first two components. Thus, $\rightarrow_M$ is a binary relation, not a function. The notion of the language accepted by a nondeterministic Turing Machine M, denoted by $L(M)$, is like the one for a deterministic Turing Machine. Thus, we have:

$$L(M) = \{w \mid w \in \Sigma^* \text{ and } q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2 \text{ for some } q \in F,\ \alpha_1 \in \Gamma^*, \text{ and } \alpha_2 \in \Gamma^+\}$$

Often one refers to the nondeterminism of a Turing Machine as being *angelic*, in the sense that $w \in L(M)$ iff *there exists* a sequence of quintuples in the transition relation δ such that $q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2$ for some $q \in F$, $\alpha_1 \in \Gamma^*$, and $\alpha_2 \in \Gamma^+$.

The deterministic Turing Machine M which simulates a given nondeterministic Turing Machine N, uses three tapes and it works as follows. (The inverse simulation is obvious because the deterministic Turing Machine is a particular case of the nondeterministic Turing Machine.)

Let us assume that the transition relation of the nondeterministic Turing Machine N has at most $b\,(\geq 2)$ quintuples with the same first two components. We also assume that the quintuples of the transition relation of N are ordered, so that it makes sense to choose the n-th quintuple, for $n = 1, \ldots, b$, among all those with the same first two components. If n is larger than the number of quintuples with the given first two components, then we say that we made an *improper choice* of the quintuple. On the first tape M keeps a copy of the input of N. On the second tape M generates in the canonical order (see Section 1 on page 9) the strings in $\{1, \ldots, b\}^*$, where $1 < \ldots < b$, that is, it generates the strings $\varepsilon, 1, \ldots, b, 11, \ldots, 1b, 21, \ldots, 2b, \ldots, b1, \ldots, bb, 111, \ldots, 11b, \ldots$, one at a time ($\varepsilon$ is the empty sequence). After every move only one string appears on the second tape. It is clear that after a finite number of moves, any given string s of k numbers, each in $\{1, \ldots, b\}$, will appear on the second tape. It is also clear that M can compute any next string of the canonical order from the previous one it finds written on the tape.

The Turing Machine M simulates the Turing Machine N by first copying on the third tape the input (which is on its first tape) and then reading from-left-to-right the sequence of numbers, each one in $\{1, \ldots, b\}$, written on the second tape as a sequence of instructions for choosing the quintuple among those available at each move. The empty sequence means that no move should be made. Suppose that the sequence is: 2 3 1 3 4. It means that on the first move, for the given scanned symbol and the given state, M should choose the 2nd quintuple among the quintuples which apply for the first move, then for the 2nd move M should choose the 3rd quintuple among the quintuples which apply for the second move, for the 3rd move M should choose the 1st quintuple among the quintuples which apply for the third move, and so on.

If the sequence on the second tape forces an improper choice then M proceeds as follows.

(1) M generates on the second tape the sequence which in the canonical order follows the sequence which is currently written on the tape.

(2) Then M copies again the input from the first tape onto the third tape and makes its moves as specified by the sequence, by reading the numbers of the sequence from-left-to-right. If the sequence has been completely exhausted and M did not enter a final state then M proceeds by first generating the next sequence on the second tape and proceeding as indicated in this Point (2).

If after any move the machine M ever enters a final state then M stops and accepts the input word.

Now, it is obvious that for any input word w, if M accepts w then there exists a sequence of choices among the various alternative for the application of the transition relation δ such that N accepts w. Since by the canonical generation, M generates in a finite number of moves any given sequence of numbers, M accepts w iff N accepts w.

In order for M to simulate N (in the sense that $L(N) = L(M)$) it is not necessary that M generates the sequences of numbers precisely in the canonical order. Other orders are possible as we now indicate.

Let us first note that a set of strings which is closed under the prefix operation, naturally defines a tree. Thus, the set of strings in $\{1, \ldots, b\}^*$ which are closed under the prefix operation, defines a tree. Let us call it T. A finite sequence of numbers in

$\{1,\ldots,b\}^*$ can be identified with a node in T. For instance, in Figure 9, the sequence 2 2 4 4 3 corresponds to the node n. Now, in order for M to simulate N it is enough that M generates the various sequences of numbers, each sequence being in $\{1,\ldots,b\}^*$, so that the generation of the corresponding nodes of the tree T satisfies the following property:

Property (A): for any already generated node r of the tree T, we have that within a finite number of moves after the generation of r, the Turing Machine M generates *all* nodes which are the leaves of a finite subtree T_r of T such that: (i) r is the root of T_r, (ii) T_r includes other nodes besides r, and (iii) if a node q in T_r is a son of a node p in T_r then *all* sons of p belong to T_r.

In order to start the process of node generation, we assume that the root node of the tree T (that is, the empty sequence ε) has already been generated by M before it starts its simulation of the nondeterministic Turing Machine N.

We leave it to the reader to prove that indeed M simulates N iff it generates the sequence of numbers in any order whose corresponding node generation complies with Property (A) above.

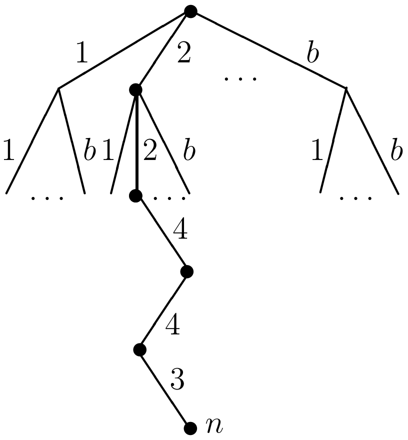


Fig. 9. The tree of choice-sequences. The node n corresponds to the sequence: 2 2 4 4 3.

Remark 7. The property that nondeterministic Turing Machines are equivalent to deterministic Turing Machines should be compared with the analogous property holding for Finite Automata. The expert reader may also recall that this is *not* the case for Pushdown Automata, that is, nondeterministic Pushdown Automata [16] accept a class of languages which is strictly larger than the class accepted by deterministic Pushdown Automata. For Linear Bounded Automata [16] (which are nondeterministic automata) it is an open problem whether or not they accept a class of languages which is strictly larger than the class accepted by deterministic Linear Bounded Automata.

5 Restrictions to Turing Machines Which Do Not Diminish Their Power

In this section we study some restrictions to the Turing Machines which do *not* diminish their computational power. We study these restrictions because they will help us to relate the Turing computable functions and the functions which are computable according to other models of computation, which we will introduce in Section 9.

5.1 One Tape = One Tape + Read-Only Input Tape = Two Stacks

Definition 11. [Deterministic Two-Stack Machine] A *deterministic two-stack machine* (or *two-stack machine*, for short) is an off-line deterministic Turing Machine with its read-only input tape and, instead of the working tape, it has two tapes which are used with the following restrictions (which make the tapes to behave like stacks):
(i) they are one-way infinite to the right, and
(ii) when the head moves to the left a printable blank is printed.

A deterministic two-stack machine is depicted in Figure 10.

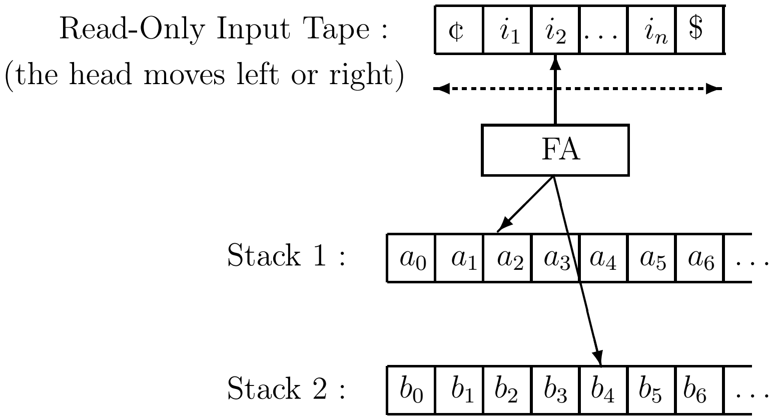


Fig. 10. A deterministic two-stack machine.

Theorem 1. A deterministic Turing Machine can be simulated by a deterministic two-stack machine.

Proof. As indicated in Figure 11, the tape configuration of a deterministic Turing Machine can be simulated by two stacks. In that figure the two stacks are α_1 and α_2. Their top elements are a_n and a_{n+1}, respectively. It is assumed that the scanned cell is the top of the stack α_2.

A left move which reads a symbol x and writes a symbol y, is simulated by: (i) popping the symbol x from the stack α_2, (ii) pushing the symbol y onto the stack α_2, (iii) popping the top, say t, of α_1, and finally, (iv) pushing t onto α_2. Analogously, a right move is simulated by the same sequence of operations, except that at Step (iii) we pop from α_2 and at Step (iv) we push onto α_1.

When the stack α_1 is empty the head is on the leftmost position. When the stack α_2 is empty the head is on the leftmost blank. The blank symbol never needs to be placed on the stack. □

Theorem 2. A deterministic two-stack machine $M2_I$ with its read-only input tape can be simulated by a deterministic two-stack machine $M2$ without a read-only input tape.

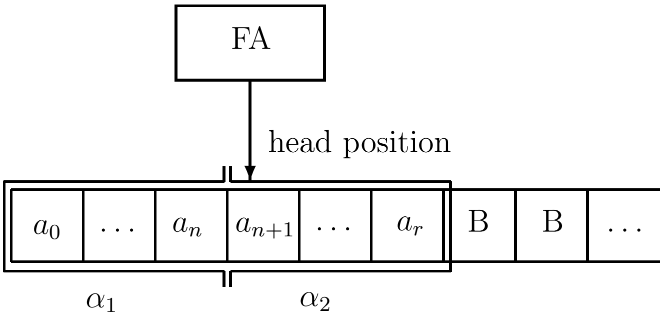


Fig. 11. Two stacks α_1 and α_2 which simulate a Turing Machine tape.

Proof. A two-stack deterministic machine $M2_I$ with a read-only input tape can be simulated by an off-line Turing Machine M with a read-only input tape and one standard tape. The machine M is equivalent to a Turing Machine $M1$ with one standard tape only, and finally, by Theorem 1, $M1$ can be simulated by a deterministic two-stack machine $M2$.

An alternative proof is as follows. Consider, as above, the off-line Turing Machine M with input tape I and standard tape T, which is equivalent to $M2_I$. Then the Turing Machine M can be simulated by a Turing Machine M_{tt} with two-track tape T_{tt}. Indeed, M_{tt} simulates M by considering both:
(i) the input symbol read by M (marked by the head position $X1$) on the leftmost, read-only part of its tape T_{tt} between the cells $\begin{vmatrix} ¢ \\ B \end{vmatrix}$ and $\begin{vmatrix} \$ \\ B \end{vmatrix}$, and
(ii) the tape symbol read by M (marked by the head position $X2$) on the part of its tape T_{tt} to the right of the cell $\begin{vmatrix} \$ \\ B \end{vmatrix}$ (see Figure 12). Finally, the Turing Machine M_{tt} with tape T_{tt} can be simulated by a two-stack machine. □

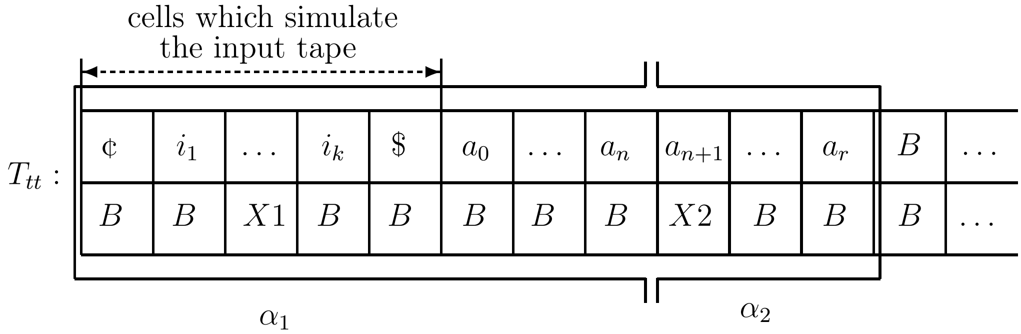


Fig. 12. Two stacks α_1 and α_2 which simulate a tape and a read-only input tape. $X1$ and $X2$ show the head position on the input tape and on the standard tape, respectively.

5.2 Two Stacks = Four Counters

Let us introduce the notion of a counter machine with n counters, which is depicted in Figure 13.

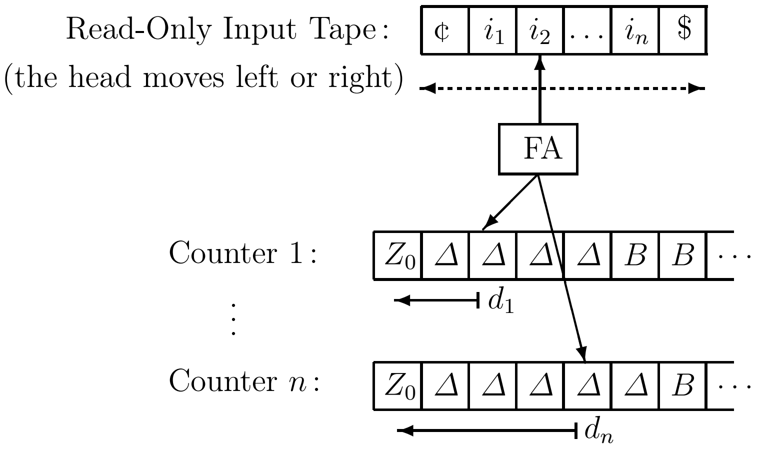


Fig. 13. A Counter Machine with n counters. As usual, the finite automaton FA is deterministic. B is the blank symbol and $\varDelta$ is a printable blank symbol. Counter 1 holds 2 and Counter n holds 4.

Definition 12. [Counter Machine] A *counter machine* with n counters, with $n>0$, is a deterministic finite automaton with a read-only input tape and n stacks. Each stack which is said to be a *counter*, has two symbols: Z_0 and $\varDelta$ (printable blank) only. The following conditions are satisfied: (i) Z_0 is initially written on the bottom of the stack, (ii) on the bottom cell only Z_0 can be written, (iii) on the other cells of the stack only $\varDelta$ can be written, and (iv) one can test whether or not a head is reading Z_0.

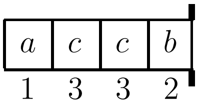
Thus, each counter of a counter machine is a device for storing an integer, say k, and the available operations are: (i) addition of 1 (this operation can be performed by writing the scanned symbol and moving to the right, then writing $\varDelta$ and moving to the left, and, finally, writing the scanned symbol and moving to the right), (ii) subtraction by 1 (this operation can be performed by writing the scanned symbol and moving to the left), and (iii) test for 0 (this operation can be performed by testing for Z_0).

A counter stores the integer k if the scanned symbol is the k-th $\varDelta$ symbol to the right of Z_0.

Actually, stacks of the form described in the above Definition 12 are called *iterated counters* (see, for instance, [14]), rather than simply counters. In [14] the name 'counter' is reserved to stacks where one can perform: (i) addition of 1, and (ii) subtraction by 1 only.

Theorem 3. A Turing Machine M can be simulated by a counter machine with four counters.

Proof. We first show how to simulate a stack using two counters. Suppose that the stack has $k-1$ tape symbols. In this case each stack configuration can be encoded by a number written in base k, as we indicate through the following example.

Let us assume that the tape symbols are a, b, and c. They can be encoded, using the base 4, by the numbers 1, 2, and 3, respectively. Now, let us consider the stack configuration $a\,c\,c\,\underset{\blacktriangle}{b}$, with top symbol b. (The symbol ▲ indicates the position of the top of the stack.) That configuration can be depicted as follows (below each symbol we have also written its encoding number): , and can be encoded by the number 1332 (in base 4), that is, $1\times4^3 + 3\times4^2 + 3\times4^1 + 2\times4^0$, which is 126 (in base 10).

▲

Note that we should use the base k and avoid the use of the digit 0 because otherwise we will identify two different stack configurations which differ by leading 0's only. For instance, if the symbols a, b, and c are encoded by 0, 1, and 2, respectively, and we use the base 3, then the tape configurations $c\,b$ and $a\,c\,b$ are encoded by the same number 7 (in base 10). Indeed, $c\,b$ is encoded by $2\times3^1 + 1\times3^0$ and $a\,c\,b$ by $0\times3^2 + 2\times3^1 + 1\times3^0$.

Thus, (i) the operation '*push x*' on the stack is simulated by the multiplication by k and the addition of $H(x)$, where $H(x)$ is the number encoding the tape symbol x, (ii) the operation '*pop*' is simulated by the integer division by k (discarding the remainder), and (iii) the operation '*top*' is simulated by copying the number encoding the current configuration, say j, into the second tape while computing the top symbol as the value of j modulo k.

The above operations can be performed as follows:
(i) addition of n on a counter can be performed by moving its head n cells to the right,
(ii) multiplication by k of a number on the counter $C1$ with the result on the counter $C2$, can be performed by going left one cell on $C1$ while going right (starting from the leftmost cell) k cells on $C2$ until $C1$ reaches the bottom cell,
(iii) in an analogous way, division by k can be done by moving left k cells on a counter while the other goes right one cell (care should be taken, because if it is impossible to go left by k cells, then the head of the other counter should not move to the right), and
(iv) while going left on one counter, say $C1$, by 1 cell, we go right on the other counter $C2$ by 1 cell and at the same time in the states of the finite automaton we move by 1 state in a circle of k states to recall j modulo k. At the end, when the copying operation is finished, the state reached will tell us the value of j modulo k which can be stored in $C1$. □

5.3 Four Counters = Two Counters

Theorem 4. A Turing Machine M can be simulated by a counter machine with two counters.

Proof. By Theorem 3 it is enough to show how four counters can be simulated by two counters, call them $D1$ and $D2$. We encode the four values of the counters, say p, q, r, s by the number $2^p\ 3^q\ 5^r\ 7^s$ to be placed on the counter $D1$. Obviously, two different 4-tuples are encoded by two different numbers. Any other choice of four prime numbers, instead of 2, 3, 5, and 7, is valid.

We need to show: (a) how to increment a counter, (b) how to decrement a counter, and (c) how to test whether or not a counter holds 0. Let $C(1) = 2$, $C(2) = 3$, $C(3) = 5$, and $C(4) = 7$. Let us assume that the given counter configuration p, q, r, s is encoded in the counter $D1$.
Point (a). To increment the i-th counter by 1 we multiply the counter $D1$ by $C(i)$. This is done by going one cell to the left in $D1$ while going $C(i)$ cells to the right on $D2$, starting

from the leftmost cell. The result will be on the counter $D2$, and if we want the result on $D1$ we have then to copy $D2$ onto $D1$.

Point (b). To decrement the i-th counter by 1 we perform an integer-division of the counter $D1$ by $C(i)$. This is done analogously to what is done at Point (a).

Point (c). To determine whether or not p or q or r or s is 0 we first compute $2^p\ 3^q\ 5^r\ 7^s$ modulo 2, as specified in the proof of Theorem 3 (recall that by that operation the value of $2^p\ 3^q\ 5^r\ 7^s$ is not destroyed) and it is 0 iff $p\neq 0$ (because $2^p\ 3^q\ 5^r\ 7^s$ is even iff $p\neq 0$). If $p=0$ then the finite control enters a suitable state, say α_2 (see Figure 14 on page 36).

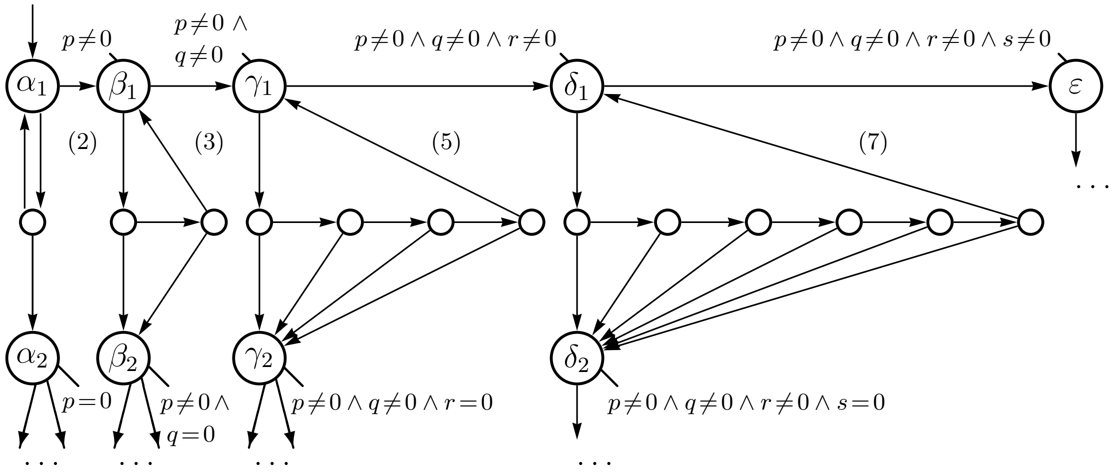


Fig. 14. The simulation of four counters by two counters. (i) In α_2: $p=0$. (ii) In β_1: $p\neq 0$. (iii) In β_2: $p\neq 0 \wedge q=0$. (iv) In γ_1: $p\neq 0 \wedge q\neq 0$. (v) In γ_2: $p\neq 0 \wedge q\neq 0 \wedge r=0$. (vi) In δ_1: $p\neq 0 \wedge q\neq 0 \wedge r\neq 0$. (vii) In δ_2: $p\neq 0 \wedge q\neq 0 \wedge r\neq 0 \wedge s=0$. (viii) In ε: $p\neq 0 \wedge q\neq 0 \wedge r\neq 0 \wedge s\neq 0$.

We may also compute $2^p\ 3^q\ 5^r\ 7^s$ modulo 3, modulo 5, and modulo 7 to check whether or not $q=0$, or $r=0$, or $s=0$, respectively. Clearly this computation can be performed by the finite control of a counter machine. If none of the numbers p, q, r, and s is 0 then the finite control of the counter machine enters the state ε (see Figure 14). In that figure the states α_1, β_1, γ_1, δ_1, and the 'small states' with no label inside, are the states visited while computing the value of $2^p\ 3^q\ 5^r\ 7^s$ modulo 2, 3, 5, and 7. This computation can easily be extended to the case when the test should return a different answer according to which of the 16 combinations of the four values of p, q, r, and s (each of these values can be either 0 or different from 0). In this case the finite control should enter one of 16 possible 'answer states'. In Figure 14 we have depicted some of the states of the finite control for testing the values of p, q, r, and s: they correspond to the tree of Figure 15 on page 37.

The 16 answer states would be the leaves of the tree of Figure 15 when the tree is completed by constructing: (i) a subtree rooted in γ_2 analogous to the one rooted in δ_1, (ii) a subtree rooted in β_2 analogous to the one rooted in γ_1, and (iii) a subtree rooted in α_2 analogous to the one rooted in β_1.

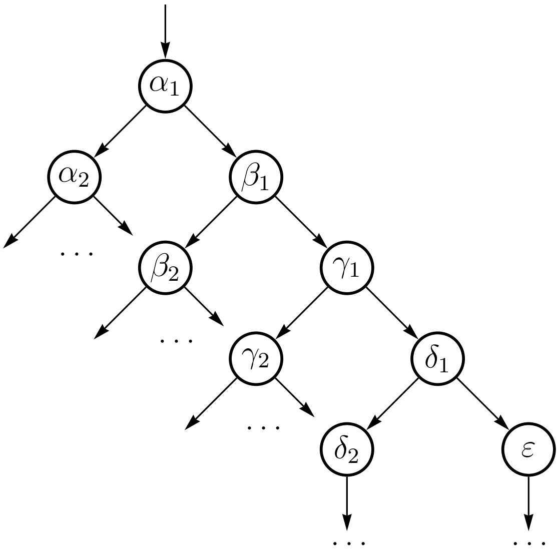


Fig. 15. A decision tree for testing the values of the four counters.

The position of the reading head on the input tape of the two-counter machine is the same as the position of the four-counter machine.

Finally, we have to show how the two-counter machine can choose the move corresponding to the move of the four-counter machine. Any move of the four-counter machine depends on: (i) its internal state, and (ii) the position of the heads of the four counters, which can be one of the following sixteen configurations: (1) all four heads pointing to Z_0, (2–5) three heads pointing to Z_0 and one pointing to $\varDelta$, (6–11) two heads pointing to Z_0 and two pointing to $\varDelta$, (12–15) one head pointing to Z_0 and three pointing to $\varDelta$, (16) all four heads pointing to $\varDelta$.

The information about the internal state of the four-counter machine and the position of the heads of the four counters (which can be determined as we have seen at Point (c) above) can be stored in the internal states of the two-counter machine. In order to store the above information, the internal states of the two-counter machine may be very large. Indeed, in general, they should replicate the states of the four-counter machine for each of the sixteen configurations of the positions of the heads of the four counters.

Then, the two-counter machine can make the move corresponding to the one of the four-counter machine by suitably modifying the content of its two counters and changing its internal state. □

5.4 Turing Machines with Three or Two States Only

We state without proof the following result [37].

Theorem 5. [Shannon 1956] Given an alphabet $\varSigma$ and a language $R \subseteq \varSigma^*$ such that $R = L(M)$ for some Turing Machine M then there exists a Turing Machine $M3$ and a tape alphabet $\varSigma3 \supseteq \varSigma$ such that $R = L(M3)$ and $M3$ has three states.

If one assumes that the acceptance of a word in Σ^* is by halting (and not by entering a final state, as we have stipulated above), then we can reduce the number of states from 3 to 2. (Actually, this is what Shannon originally proved.)

5.5 Turing Machines with Three or Two Tape Symbols Only

Theorem 6. Given a language $R \subseteq \{0,1\}^*$ such that $R = L(M)$ for some Turing Machine M then there exists a Turing Machine $M3$ which uses the tape alphabet $\{0,1,B\}$, such that $R = L(M3)$.

Proof. Every tape symbol for the machine M will be encoded by a sequence of 0's and 1's of length k, where k is known when M is given. Let us assume that $M3$ is reading the leftmost cell of the k cells which encode the symbol read by M. $M3$ simulates the move of M by first reading k symbols to the right. If M reads a blank symbol then $M3$ should first write the code for the blank symbol on its tape (using k symbols to the right of the current position). The machine $M3$ begins the simulation of the machine M by first computing the encoding $en(w)$ of the word w which is the input of M. That encoding is computed starting from the leftmost symbol of w. During that computation the symbols of w to the right of the symbol, say s, which will be encoded, are shifted to the right by $k-1$ cells to create k free cells for the code of the symbol s itself.

In contrast to what is done for the input the encoding of the blank symbols is not done initially, because there are infinite blanks. The encoding of one blank symbol is done on demand, one blank at a time when it is required. □

Corollary 1. Given a language $R \subseteq \Sigma^*$ for any finite input alphabet Σ, if $R = L(M)$ for some Turing Machine M then there exists an off-line Turing Machine $M3$ with the input tape and a standard tape whose alphabet is $\{0,1,B\}$ such that $R = L(M3)$.

Turing Machines whose input alphabet is $\{0,1\}$ and tape alphabet is $\{0,1,B\}$, are also called *binary Turing Machines*.

Theorem 7. [Wang 1957] Given a language $R \subseteq \Sigma^*$ for any finite input alphabet Σ, if $R = L(M)$ for some Turing Machine M then there exists an off-line Turing Machine $M2$ with: (i) an read-only input tape whose alphabet is Σ, and (ii) a standard tape whose alphabet is $\{0,1\}$, where 0 plays the role of blank (thus, the blank symbol B does not exist in the tape alphabet), such that $R = L(M2)$ and $M2$ cannot write a 0 over a 1 (that is, $M2$ cannot erase symbols: $M2$ has a non-printable blank symbol 0).

Proof. The simulation of M by $M2$ proceeds by rewriting along the tape the configurations of M. We need special symbols (written as strings of 0's and 1's) to encode the beginning and the end of each configuration, the position of the head, and some suitable markers to guide the process of copying, with the necessary changes, the old configuration into the new one. □

6 Turing Machines as Language Generators

Let us consider a fixed alphabet Σ. A Turing Machine can be viewed as a mechanism for generating languages that are subsets of Σ^* as follows. We consider a two-tape Turing

Machine M. The first tape is one-way infinite and it is used by the Turing Machine M as usual. The second tape is an *output tape*, that is, it is a write-only tape whose head moves to the right only. On this second tape the machine M prints, one after the other, separated by a special symbol, say #, not in Σ, the words of the language it generates.

The language $L \subseteq \Sigma^*$ generated by a Turing Machine M will be denoted by $G(M)$. If $G(M) \subseteq \Sigma^*$ is infinite then M does not halt.

Note that we do not require that on the output tape the words of $G(M)$ are generated either (i) only once, or (ii) in any fixed order. In particular, for any given word $w \in G(M)$ of length $|w|$ (≥ 0) and of the form: $x_1 x_2 \ldots x_{|w|}$, there exists an index $i \in N$, with $i \geq 0$, such that the output tape cells $c_i, c_{i+1}, \ldots, c_{|w|-1}, c_{|w|}$ contain the symbols $x_1, x_2, \ldots, x_{|w|}, \#$, respectively.

Theorem 8. For any given Turing Machine M and language R, $R = G(M)$ iff there exists a Turing Machine $M1$ such that $R = L(M1)$, that is, a language is generated by a Turing Machine M iff there exists a Turing Machine $M1$ which accepts it (for the notion of the language accepted by a Turing Machine see Definition 8 on page 14).

Proof. (*only-if part*) Let us assume that R is generated by the machine M. The machine $M1$ which accepts R can be constructed as follows. Given an input word $w \in \Sigma^*$, $M1$ works as M and when M writes a word $w_1\#$ on the output tape, $M1$ compares w_1 with w. If $w_1 = w$ then $M1$ accepts w, otherwise $M1$ continues the simulation of M. Obviously, $M1$ accepts w iff w is generated by M.

(*if part*) Given a Turing Machine $M1$ we have to construct a Turing Machine M which generates exactly the same language accepted by $M1$. Before showing this construction we need to define the notion of the i-th word in Σ^*, for any given finite alphabet Σ. (Actually this notion can be extended to the case where Σ is denumerable.) We can use the canonical order of the strings in Σ^*. In particular, if $\Sigma = \{0, 1\}$ and we assume that $0 < 1$, then the i-th word of Σ^* is the i-th word occurring in the listing of the *canonical* order of the words of Σ^*, that is: $\varepsilon, 0, 1, 00, 01,\ 10, 11, 000,\ 001, 010,\ 011, 111, \ldots$. Thus, the i-th word of Σ^* is the i-th word in the listing Σ^0, Σ^1, Σ^2, ..., having listed the elements within each set Σ^i (for $i \geq 0$) in the usual *alphabetic* order from left to right. Obviously, one can devise a Turing Machine for computing the i-th word for any given $i \geq 0$. The machine M has three tapes plus an output tape. On the first tape it generates all natural numbers in increasing order starting from 0. For each such number, say n, M produces on the second tape a pair of numbers, say $\langle i, j\rangle$, using a dove-tailing bijection from $N \times N$ to N. Then given i, the Turing Machine M computes on the third tape the i-th word, say w_i, of Σ^*. At the end of this computation the third tape contains the word w_i. Then M behaves on its third tape as $M1$ does on its own tape but only for j steps (that is, j applications of the transition function), and if $M1$ accepts w_i using *no more than* j steps then M writes on its output tape w_i. □

Theorem 9. A language R is generated by a Turing Machine M, that is, $R = G(M)$, and each word in R is generated *once only*, iff R is accepted by some Turing Machine $M1$, that is, $R = L(M1)$. The language R is recursive enumerable (see Definition 1 on page 87 and also Section 15 on page 79).

Proof. The proof is like the one of Theorem 8, where, instead of 'no more than j', we say 'exactly j', because if a language is accepted by a Turing Machine there exists a deterministic Turing Machine which accepts it, and if a deterministic Turing Machine $M1$ accepts a word, say w, then $M1$ accepts w in exactly one way, that is, after a number of steps which depends on w only. □

Theorem 10. A language R is generated by a Turing Machine M, that is, $R = G(M)$, and it is generated in the *canonical order* iff there exists a Turing Machine $M1$ which accepts R, that is, $R = L(M1)$, such that for all $w \in \Sigma^*$, $M1$ stops after a finite number of moves (possibly depending on w). The language R is recursive (see Definition 2 on page 87).

Proof. (*only-if part*) The machine $M1$ to be constructed, simulates M until: either (i) M halts without generating w, in which case $M1$ rejects w, or (ii) w is generated by M, in which case $M1$ accepts w, or else (iii) a word longer than w is generated by M (and this means that w will not be generated in the future by M, being the generation done by M in canonical order), in which case $M1$ rejects w.

(*if part*) Given the language R accepted by the machine $M1$, that is, $R = L(M1)$, we construct the machine M which generates R in the canonical order, as follows. If R is finite M looks at the elements of R and rearrange them in the canonical order and output them. If R is infinite, the language itself is given via the Turing Machine $M1$, and the machine M to be constructed, works as follows. M computes each word w in Σ^* in the canonical order and for each of them, M (i) simulates $M1$ and (ii) generates the word w of R iff $M1$ accepts w. Since for every input word w in Σ^*, $M1$ accepts or rejects w after a finite number of moves, we have that for any given word w in R, M generates w after a finite number of moves. It is immediate to see that the machine M indeed generates the language R in the canonical order. □

7 Turing Computable Functions and Type 0 Languages

In this section we state and prove the equivalence between Turing Machines and type 0 languages.

Theorem 11. [Equivalence Between Type 0 Grammars and Turing Machines. Part 1] For any language $R \subseteq \Sigma^*$ if R is generated by the type 0 grammar $G = \langle \Sigma, V_N, P, S \rangle$, where Σ is the set of terminal symbols, V_N is the set of nonterminal symbols, P is the finite set of productions, and S is the axiom, then there exists a Turing Machine M such that $L(M) = R$.

Proof. Given the grammar G which generates the language R, we construct a nondeterministic Turing Machine M with two tapes as follows. Initially, on the first tape there is the word w to be accepted iff $w \in R$, and on the second tape there is the sentential form consisting of the axiom S only. Then M simulates a derivation step of w from S by performing the following Steps (1), (2), and (3). *Step* (1): M chooses in a nondeterministic way a production of the grammar G, say $\alpha \rightarrow \beta$, and an occurrence of α on the second tape. *Step* (2): M rewrites that occurrence of α by β, thereby changing the string on the second tape. In order to perform this rewriting, M may apply the *shifting-over* technique for Turing Machines [16] by either shifting to the right if $|\alpha| < |\beta|$, or shifting to the left if $|\alpha| > |\beta|$. *Step* (3): M checks whether or not the string produced on the second tape is equal to the word w which is kept unchanged on the first tape. If this is the case, M accepts w and stops. If this is not the case, M simulates one more derivation step of w from S by performing again Steps (1), (2), and (3) above.

We have that $w \in R$ iff $w \in L(M)$. □

Theorem 12. [Equivalence Between Type 0 Grammars and Turing Machines. Part 2] For any language $R \subseteq \Sigma^*$ such that there exists a Turing Machine M such that $L(M) = R$ then there exists a type 0 grammar $G = \langle \Sigma, V_N, P, A_1 \rangle$, where Σ is the set of terminal symbols, V_N is the set of nonterminal symbols, P is the finite set of productions, and A_1 is the axiom, such that R is the language generated by G.

Proof. Given the Turing Machine M and a word $w \in \Sigma^*$, we construct a type 0 grammar G which first makes two copies of w and then simulates the behaviour of M on one copy. If M accepts w then $w \in L(G)$, and if M does not accept w then $w \notin L(G)$. The detailed construction of G is as follows.
Let $M = \langle Q, \Sigma, \Gamma, q_0, B, F, \delta \rangle$. The productions of G are the following ones, where the pairs of the form $[-,-]$ are elements of the set V_N of the nonterminal symbols:

1. $A_1 \rightarrow q_0\ A_2$

The following productions nondeterministically generate two copies of w:

2. $A_2 \rightarrow [a, a]\ A_2$ for each $a \in \Sigma$

The following productions generate all tape cells necessary for simulating the computation of the Turing Machine M:

3.1 $A_2 \rightarrow [\varepsilon, B]\ A_2$
3.2 $A_2 \rightarrow [\varepsilon, B]$

The following productions simulate the moves to the right:

4. $q\,[a, X] \rightarrow [a, Y]\,p$
   for each $a \in \Sigma \cup \{\varepsilon\}$,
   for each $p, q \in Q$,
   for each $X \in \Gamma$, $Y \in \Gamma - \{B\}$ such that $\delta(q, X) = (p, Y, R)$

The following productions simulate the moves to the left:

5. $[b, Z]\,q\,[a, X] \rightarrow p\,[b, Z]\,[a, Y]$
   for each $a, b \in \Sigma \cup \{\varepsilon\}$,
   for each $p, q \in Q$,
   for each $X, Z \in \Gamma$, $Y \in \Gamma - \{B\}$ such that $\delta(q, X) = (p, Y, L)$

When a final state q is reached, the following productions propagate the state q to the left and to the right, and generate the word w, making q to disappear when all the terminal symbols of w have been generated:

6.1 $[a, X]\,q \rightarrow q\,a\,q$
6.2 $q\,[a, X] \rightarrow q\,a\,q$
6.3 $q \rightarrow \varepsilon$

for each $a \in \varSigma \cup \{\varepsilon\}$, $X \in \varGamma$, $q \in F$

We will not formally prove that all the above productions simulate the behaviour of M, that is, for any $w \in \varSigma^*$, $w \in L(G)$ iff $w \in L(M)$.

The following observations should be sufficient:

(i) the first components of the nonterminal symbols $[-,-]$ are never touched by the productions so that the given word w is kept unchanged,

(ii) never a nonterminal symbol $[-,-]$ is made to be a terminal symbol if a final state q is not encountered first,

(iii) if the acceptance of a word w requires at most $k\,(\geq 0)$ tape cells, we have that the initial configuration of the Turing Machine M for the word $w = a_1\,a_2 \ldots a_n$, with $n \geq 0$, on the leftmost cells of the tape, is simulated by the derivation:

$$A_1 \;\rightarrow^*\; q_0\,[a_1,a_1]\,[a_2,a_2]\ldots[a_n,a_n]\,[\varepsilon,B]\,[\varepsilon,B]\ldots[\varepsilon,B]$$

where there are k $(\geq n)$ nonterminal symbols to the right of q_0. □

8 Multitape Turing Machines and Random Access Memory Machines

A RAM (*Random Access Machine*) is an abstract model of computation which consists of an infinite number of memory cells (which are indexed by the numbers $0, 1, \ldots$), each of which can hold an integer, and a finite number of arithmetic registers, capable of storing and manipulating integers according to some given set of elementary instructions such as: STORE, LOAD, ADD, SUB (that is, subtract), MULTIPLY, DIV (that is, divide), JUMP, CONDITIONAL-JUMP (that is, jump if the content of some given register is 0), and STOP.

Definition 13. [Computable Function From N To N] A function f from N to N is said to be *RAM computable* iff starting from n in memory cell 0 we get the value of $f(n)$ in memory cell 0 when executing the instruction STOP.

As usual, integers may encode instructions. The instructions, once interpreted, modify the content of the memory cells and/or the registers.

It is not difficult to convince ourselves that a RAM can simulate any given Turing Machine or digital computer.

Now we show that, provided that the elementary instructions can be simulated by a Turing Machine (and this is left to the reader as an exercise, by using the techniques for the construction of Turing Machines that we have presented in previous sections), a RAM R can be simulated by a Turing Machine M. (In what follows, for simplicity, we will not distinguish between numbers and their encoding representations on the four tapes in use.) We assume that R has k registers.

Let us consider the following Turing Machine M with four tapes:

memory tape (n cells):	$\#\,0 \,*\, v_0\,\#\,1 \,*\, v_1\,\#\ldots\#\,i \,*\, v_i\,\#\ldots\#\,n \,*\, v_n\,\#\,B\,B\ldots$
register tape (k registers):	$\#\,0 \,*\, v_0\,\#\,1 \,*\, v_1\,\#\ldots\#\,k \,*\, v_k\,\#\,B\,B\ldots$
program counter:	$\#\,i\,\#\,B\,B\ldots$ (i is a memory address)
memory address register:	$\#\,j\,\#\,B\,B\ldots$ (j is a memory address)

If the program counter holds the value i then M looks for the i-th memory cell in the memory tape. If M does not find the string $\# i * v_i \#$ (because in the scan from the left of the memory tape it finds a B before $\# i * v_i \#$), then M stops.

Otherwise, M having found the string $\# i * v_i \#$, takes v_i and interprets it as the code of an instruction, say 'ADD j, 2' (that is, add to the register 2 the value in the memory cell whose address is j). Then M increases by 1 the program counter, looks on the memory tape for the cell whose address is j, and looks on the register tape for the register 2.

If M finds the memory cell with address j then it performs the elementary instruction of adding the number v_j to the register 2. If M does not find the memory cell with address j we may assume that the j-th cell contains 0, thus nothing should be done to the register 2. (In this case we could also assume that an error occurs and both M and R stop.) After simulating the instruction 'ADD j, 2', the machine M continues the simulation by performing the next instruction because the program counter has been already increased by 1.

As a consequence of the fact that Turing Machines can simulate RAM's and vice versa we have that the set of Turing computable functions is the same as the set of RAM computable functions.

9 Other Models of Computation

In this section we consider various other models of computation. They are all equivalent to Turing Machines as we will indicate below.

9.1 Post Machines

A *Post Machine* over the alphabet $\Sigma = \{a, b\}$ with the auxiliary symbol #, can be viewed as a flowchart whose statements operate on a queue. That queue is represented as a list $x = [x_1, \ldots, x_n]$ with the operations *head*, *tail*, *cons*, and *append* (denoted by the infix operator @) which are defined as usual. The empty list is denoted by *nil* (see also Figure 16).

In the flowchart of a Post Machine P we have the following statements:

(i) one *start* statement (with indegree 0 and outdegree 1),

(ii) 0 or more *accept* or *reject* statements (with indegree 1 and outdegree 0),

(iii) some *enqueue* statements (called *assignment* statements in [25, page 24]) (with indegree 1 and outdegree 1) of the form $enqueue(x, e)$, for $e \in \{a, b, \#\}$, which given the queue represented by the list x, produces the queue represented by the list $x@[e]$, and

(iv) some *dequeue* statements (called *test* statements in [25, page 24]) (with indegree 1 and outdegree 4) is of the form $dequeue(x)[\eta, \alpha, \beta, \chi]$ which given the queue represented by the list x, behaves as follows:

```
if $x = nil$ then goto $\eta$ else begin $h = head(x)$; $x = tail(x)$;
                                    if $h = a$ then goto $\alpha$ else
                                    if $h = b$ then goto $\beta$ else
                                    if $h = \#$ then goto $\chi$
                               end
```

where η, α, β, and χ are labels associated with some (not necessarily distinct) statements of the flowchart of the Post Machine P (see Figure 16). Every *goto* statement is an arc of the flowchart. Every *dequeue* statement includes a test on the value of *head*(x), if the list x is not *nil*, that is, if the queue is not empty.

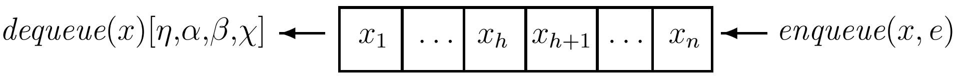


Fig. 16. The *enqueue*(x, e) and *dequeue*$(x)[\eta,\alpha,\beta,\chi]$ statements on a queue represented by the list $x = [x_1, \ldots, x_n$. We have that: *head*$(x) = x_1$ and *tail*$(x) = [x_2, \ldots, x_n]$.

The language $L \subseteq \Sigma^*$ accepted (or rejected) by a Post Machine P is the set of words w such that if the queue x is equal to w before the execution of the flowchart of P, then eventually the Post Machine P executes an *accept* statement (or a *reject* statement, respectively).

For some initial value of x, a Post Machine may run forever, without executing any *accept* or *reject* statement.

There is no loss of generality if we assume that the Post Machine has the symbols a, b, and # only. Indeed, if there are extra symbols, besides a, b, and #, we may encode them by sequences of a's and b's. For instance, if a Post Machine P' has the extra symbol Δ, then we may encode the symbol a by the string $a\,a$ and, similarly, b by $a\,b$, # by $b\,a$, and Δ by $b\,b$. Obviously, a Post Machine that uses that encoding, has to execute two dequeue statements (or two enqueue statements) for simulating a single dequeue statement (or a single enqueue statement, respectively) of the Post Machine P'.

Now we will prove the equivalence between Turing Machines and Post Machines, that is, we will prove that: (1) for any Turing Machine there exists a Post Machine which accepts the same language, and (2) vice versa.

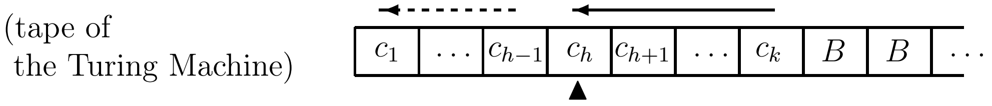
Without loss of generality, we will assume that:

(i) the tape alphabet of the Turing Machine is $\Gamma = \{a, b, B\}$, where B is the blank symbol which is not printable,

(ii) the Turing Machine cannot make a left move when its tape head scans the leftmost cell (recall that the position of the tape head can be tested by simulating a tape with two tracks), and

(iii) the Post Machine has symbols a,b, and # only.

Proof of Point (1). Let us consider a tape configuration of the Turing Machine of the form:



where the rightmost cell with a non-blank symbol is c_k and the scanned cell (marked by ▲) is c_h, with $1 \leq h \leq k$. That tape configuration is represented by the following queue q of the Post Machine:

(queue q of
the Post Machine) $$\longleftarrow \begin{array}{|c|c|c|c|c|c|c|c|} \multicolumn{4}{c}{\longleftarrow\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-} & \multicolumn{1}{c}{} & \multicolumn{3}{c}{\dashleftarrow\,\text{- - - -}} \\ \hline c_h & c_{h+1} & \dots & c_k & \# & c_1 & \dots & c_{h-1} \\ \hline \end{array} \longleftarrow \qquad (\dagger)$$

Thus, (i) the tape head of the Turing Machine scans the *leftmost cell* of the tape iff # is the rightmost element of the queue of the Post Machine (that is, the cells $c_1, \ldots, c_{h-1}$ are absent), and (ii) the tape head of the Turing Machine scans the *leftmost blank symbol B* iff # is the leftmost element of the queue of the Post Machine (that is, the cells $c_h, c_{h+1}, \ldots, c_k$ are absent). In Case (ii) if the tape of the Turing Machine has the non-blank symbols in its leftmost cells $c_1, \ldots, c_{h-1}$, then the queue of the Post Machine is of the form:

$$\longleftarrow \begin{array}{|c|c|c|c|} \hline \# & c_1 & \dots & c_{h-1} \\ \hline \end{array} \longleftarrow$$

(Here we slightly depart from the representation of the tape of the Turing Machine used in [25], but no problem arises because we assumed that when the tape head of the Turing Machine scans the leftmost cell, it cannot make a left move.) In particular, if the tape of the Turing Machine has blank symbols only and the tape head scans the leftmost blank symbol B, then the queue of the Post Machine is of the form:

$$\longleftarrow \begin{array}{|c|} \hline \# \\ \hline \end{array} \longleftarrow$$

Recall that, since the blank symbol B is not printable, the tape head of the Turing Machine scans the leftmost cell with the blank symbol B, only when the tape has blank symbols only.

We have that, before and after the simulation of a move of the Turing Machine, the queue q of the Post Machine is not empty and it has at least the element #.

Starting from the above queue q (see queue (†) on page 44), the move of the Turing Machine that writes $\widetilde{c_h}$ in the place of c_h (here and in what follows, when no confusion arises, we identify a cell with the symbol written in that cell) and goes to the right, transforms q into the following queue:

(after a right move
from queue q (†)) $$\longleftarrow \begin{array}{|c|c|c|c|c|c|c|c|} \hline c_{h+1} & \dots & c_k & \# & c_1 & \dots & c_{h-1} & \widetilde{c_h} \\ \hline \end{array} \longleftarrow$$

Analogously, the move of the Turing Machine which writes $\widetilde{c_h}$ in the place of c_h and goes to the left, transforms the above queue q (see queue (†) on page 44) into the following queue:

(after a left move
from queue q (†)) $$\longleftarrow \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline c_{h-1} & \widetilde{c_h} & c_{h+1} & \dots & c_k & \# & c_1 & \dots & c_{h-2} \\ \hline \end{array} \longleftarrow$$

Now we show how the Post Machine may simulate via fragments of flowcharts both the right move and the left move of the Turing Machine. Thus, by suitably composing those fragments, according to the definition of the transition function of the Turing Machine, we get the simulation of the Turing Machine by a Post Machine.

In Figure 17 on page 46 we show some left and right moves of the Turing Machine and the corresponding queue configurations of the Post Machine that simulates the Turing

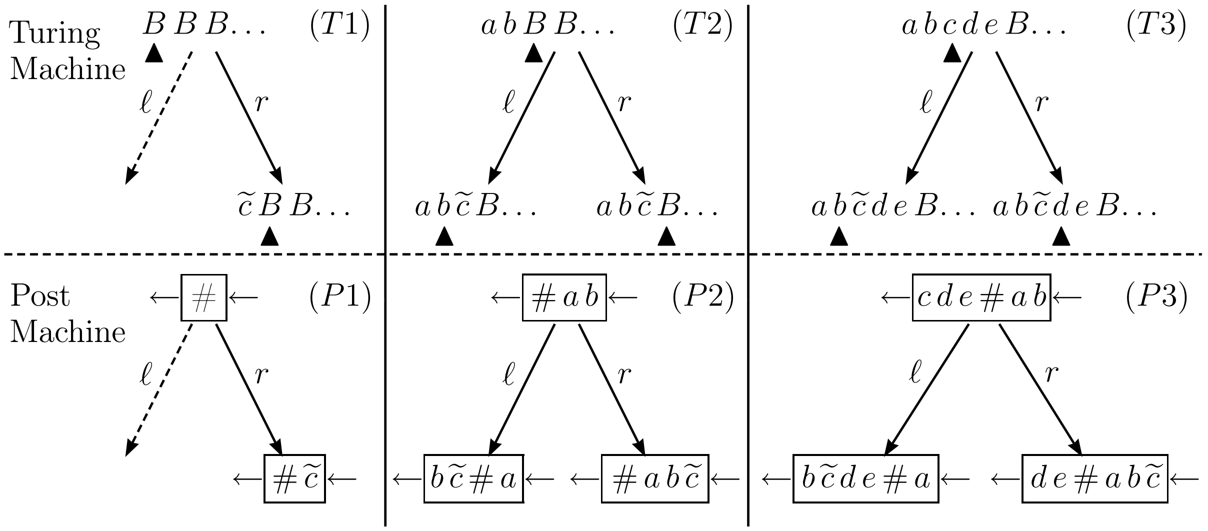

Turing Machine
$B\,B\,B\ldots$ (T1)
ℓ
r
$\widetilde{c}\,B\,B\ldots$
$a\,b\,B\,B\ldots$ (T2)
$a\,b\,\widetilde{c}\,B\ldots$
$a\,b\,\widetilde{c}\,B\ldots$
$a\,b\,c\,d\,e\,B\ldots$ (T3)
$a\,b\,\widetilde{c}\,d\,e\,B\ldots$
$a\,b\,\widetilde{c}\,d\,e\,B\ldots$
Post Machine
$\leftarrow\boxed{\#}\leftarrow$ (P1)
$\leftarrow\boxed{\#\,\widetilde{c}}\leftarrow$
$\leftarrow\boxed{\#\,a\,b}\leftarrow$ (P2)
$\leftarrow\boxed{b\,\widetilde{c}\,\#\,a}\leftarrow$
$\leftarrow\boxed{\#\,a\,b\,\widetilde{c}}\leftarrow$
$\leftarrow\boxed{c\,d\,e\,\#\,a\,b}\leftarrow$ (P3)
$\leftarrow\boxed{b\,\widetilde{c}\,d\,e\,\#\,a}\leftarrow$
$\leftarrow\boxed{d\,e\,\#\,a\,b\,\widetilde{c}}\leftarrow$

Fig. 17. Simulation of the Turing Machine by the Post Machine. Arcs with labels ℓ and r denote the moves to the left and to the right, respectively. The ▲ symbol shows the scanned cell. The symbol $\widetilde{c}$ is written in the scanned cell before moving to the left or to the right. The dashed arcs are not followed. The moves of the Turing Machine from the configurations (Ti)'s (for $i\!=\!1,2,3$) are simulated by the moves of the Post Machine from the corresponding configurations (Pi)'s.

Machine. This figure will help the reader to understand the statements of the simulating Post Machine. Here is the simulation of the right move and the left move.

Simulation of the right move. The right move of the Turing Machine that writes $\widetilde{c}$ before moving, is simulated by the Post Machine by performing the following operations:

(i) dequeue;

(ii) *if* the dequeued element is # *then* insert the element # into the queue from its left end (this insertion is performed when the queue, which may or may not be empty, has no element #);

(iii) enqueue of $\widetilde{c}$.

Thus, formally, we have that the right move is simulated by:

```
    $dequeue(q)[\eta,\alpha,\alpha,\chi]$;
$\chi$: $enqueue(q,\#)$;   $enqueue(q,\#)$;
$\omega$: $dequeue(q)[\eta_1,\alpha_1,\beta_1,\alpha]$;  $\alpha_1$: $enqueue(q,a)$;  $goto\ \omega$;  $\beta_1$: $enqueue(q,b)$;  $goto\ \omega$;
$\alpha$: $enqueue(q,\widetilde{c})$;
```

Note that there is no need for the statements with label η or η_1 because the control never goes to those statements.

Simulation of the left move. The left move of the Turing Machine that writes $\widetilde{c}$ before moving, is simulated by the Post Machine by performing the following operations (we used curly brackets for grouping operations):

(i) dequeue;

(ii) *if* the dequeued element is # *then* insert the element # into the queue from its left end;

(iii) *ℓ-enqueue* the element $\widetilde{c}$; *r-dequeue* one element, call it e; *ℓ-enqueue* the element e (††)

where *ℓ-enqueue*(q, e) denotes the insertion of the element e to the left of the queue q, and *r-dequeue*$(q)[\eta, \alpha, \beta, \chi]$ denotes the extraction of the rightmost element from the queue q. The labels η, α, β, and χ are the labels of the statements which are performed next if the queue q is empty, or the extracted element from the right is a, or b, or #, respectively. We will define the *ℓ-enqueue* and the *r-dequeue* operations below.

The operations of the *else* branch (see (††) above) are performed when in the queue there is the element # and the element e is either a or b. Note that e cannot be # because the left move is not allowed when the tape head of the Turing Machine scans the leftmost cell and only in that case the rightmost element of the queue is #.

Thus, formally, we have that the left move is simulated by:

```
           $dequeue(q)[\eta, \alpha, \alpha, \chi]$;
    $\chi$: $enqueue(q, \#)$;      $enqueue(q, \#)$;
  $\omega$: $dequeue(q)[\eta_1, \alpha_1, \beta_1, \alpha]$;  $\alpha_1$: $enqueue(q, a)$;  $goto\ \omega$;  $\beta_1$: $enqueue(q, b)$;  $goto\ \omega$;
  $\alpha$: $\ell\text{-}enqueue(q, \widetilde{c})$;
           $r\text{-}dequeue(q)[\eta_1, \alpha_1, \beta_1, \chi_1]$;
$\alpha_1$: $\ell\text{-}enqueue(q, a)$;  $goto\ \varphi$;
 $\beta_1$: $\ell\text{-}enqueue(q, b)$;  $goto\ \varphi$;
```

where the next statement to be executed has label φ. Note that there is no need for the statements with label η or η_1 or χ_1.

Here is the definition of the *ℓ-enqueue*(q, e) operation. We assume that in the queue there is the element # and the element e is either a or b. We realize the *ℓ-enqueue*(q, e) operation as follows: we insert a second element # in the queue, then we insert the element e, and finally, we make a complete rotation of the elements of the queue, but we do not reinsert to the right the second element #. Thus, we have that:

ℓ-enqueue$(q, e) =$

```
 $\lambda$: $enqueue(q, \#)$;  $enqueue(q, e)$;
 $\omega$: $dequeue(q)[\eta, \alpha, \beta, \chi]$;    $\alpha$: $enqueue(q, a)$; $goto\ \omega$;    $\beta$: $enqueue(q, b)$; $goto\ \omega$;
   $\chi$: $enqueue(q, \#)$;
$\omega_1$: $dequeue(q)[\eta_1, \alpha_1, \beta_1, \chi_1]$;   $\alpha_1$: $enqueue(q, a)$; $goto\ \omega_1$;   $\beta_1$: $enqueue(q, b)$; $goto\ \omega_1$;
```

where χ_1 is the label of the statement to be executed next. Note that there is no need for the statements with label η or η_1. Note also that at label χ_1 the second element # that was inserted in the queue q by the statement at label λ, is deleted, and the element e occurs as the first (leftmost) element of the queue q, as desired.

Here is the definition of the *r-dequeue*$(q)[\eta, \alpha, \beta, \chi]$ operation. We assume that in the queue there is the element # which is not in the rightmost position (recall that the left move cannot take place when the tape head scans the leftmost cell), and the dequeued element is a or b. We realize the *r-dequeue*$(q)[\eta, \alpha, \beta, \chi]$ operation as follows. We insert a second element # in the queue and then we make a complete rotation of the elements of the queue, reinserting to the right neither the second element # nor the element which was in the queue *to the left* of that second element #. This requires a delay of the reinsertion for checking whether or not the second element # follows the present element. Thus, we have that (see also Figure 18):

```
$r\text{-}dequeue(q)[\eta, \alpha, \beta, \chi] =$
    $\lambda$: $enqueue(q, \#)$;
    $\omega_0$: $dequeue(q)[\eta_0, \alpha_0, \beta_0, \chi_0]$;
    $\alpha_0$: $enqueue(q, a)$;  $goto\ \omega_0$;    $\beta_0$: $enqueue(q, b)$;  $goto\ \omega_0$;    $\chi_0$: $enqueue(q, \#)$;
        $dequeue(q)[\eta_1, \alpha_1, \beta_1, \chi]$;
    $\alpha_1$: $dequeue(q)[\alpha_\eta, \alpha_a, \alpha_b, \alpha]$;
    $\alpha_a$: $enqueue(q, a)$;  $goto\ \alpha_1$;    $\alpha_b$: $enqueue(q, a)$;  $goto\ \beta_1$;
    $\beta_1$: $dequeue(q)[\beta_\eta, \beta_a, \beta_b, \beta]$;
    $\beta_a$: $enqueue(q, b)$;  $goto\ \alpha_1$;    $\beta_b$: $enqueue(q, b)$;  $goto\ \beta_1$;
```

Note that there is no need for statements with label η or η_0 or η_1 or α_η or β_η or χ (recall that: (i) we have assumed that the given queue q is not empty, (ii) at label λ we have inserted in the queue q a second element #, and (iii) the first element # is not in the rightmost position).

Now let us introduce the following definition.

We say that a function f from N to N is *computable* by a Post Machine P iff starting with the string $a^n\, b\, a^{f(n)}$ in the queue, P eventually performs an *accept* statement.

Thus, as a consequence of the equivalence between Turing Machines and Post Machines which preserves the accepted language, we have that a function f from N to N is Turing computable iff f is computable by a Post Machine.

9.2 Post Systems

Given a set V of symbols, Post considered rewriting rules of the following kind to generate new strings of V^* from old strings of V^*:

$$g_0\, S_1\, g_1\, S_2\, g_2\, \ldots S_m\, g_m \;\rightarrow\; h_0\, S_{j_1}\, h_1\, S_{j_2}\, h_2 \ldots\, S_{j_n}\, h_n \qquad (\dagger)$$

where:

- $g_0, g_1, \ldots, g_m, h_0, h_1, \ldots, h_n$ are (possibly empty) strings of V^* (they are called the *fixed strings*),
- $S_1, S_2, \ldots, S_m$ are distinct symbols not in V, called *variables*, ranging over strings in V^*,
- each variable in the list $\langle S_{j_1}, S_{j_2}, \ldots, S_{j_n}\rangle$ is taken from the set $\{S_1, S_2, \ldots, S_m\}$ and need not be distinct from the other variables of the list $\langle S_{j_1}, S_{j_2}, \ldots, S_{j_n}\rangle$.

A rule of the form (†) can be used for defining a binary relation → in $V^* \times V^*$ as follows.

Given a string s in V^* we match the entire s with the whole left hand side of (†) and we get a substitution ϑ of the variables $S_1, S_2, \ldots, S_m$, each S_i, with $1 \leq i \leq m$, being mapped into a string of V^*. We then derive the rewriting:

$$s \rightarrow (h_0\, S_{j_1}\, h_1\, S_{j_2}\, h_2 \ldots S_{j_n} h_n)\vartheta.$$

For instance, given $V = \{r, s\}$ and the rule: $r\, S_1\, s\, S_2 \;\rightarrow\; S_2\, r\, S_2\, r$, we get the following rewritings of strings in V^*:

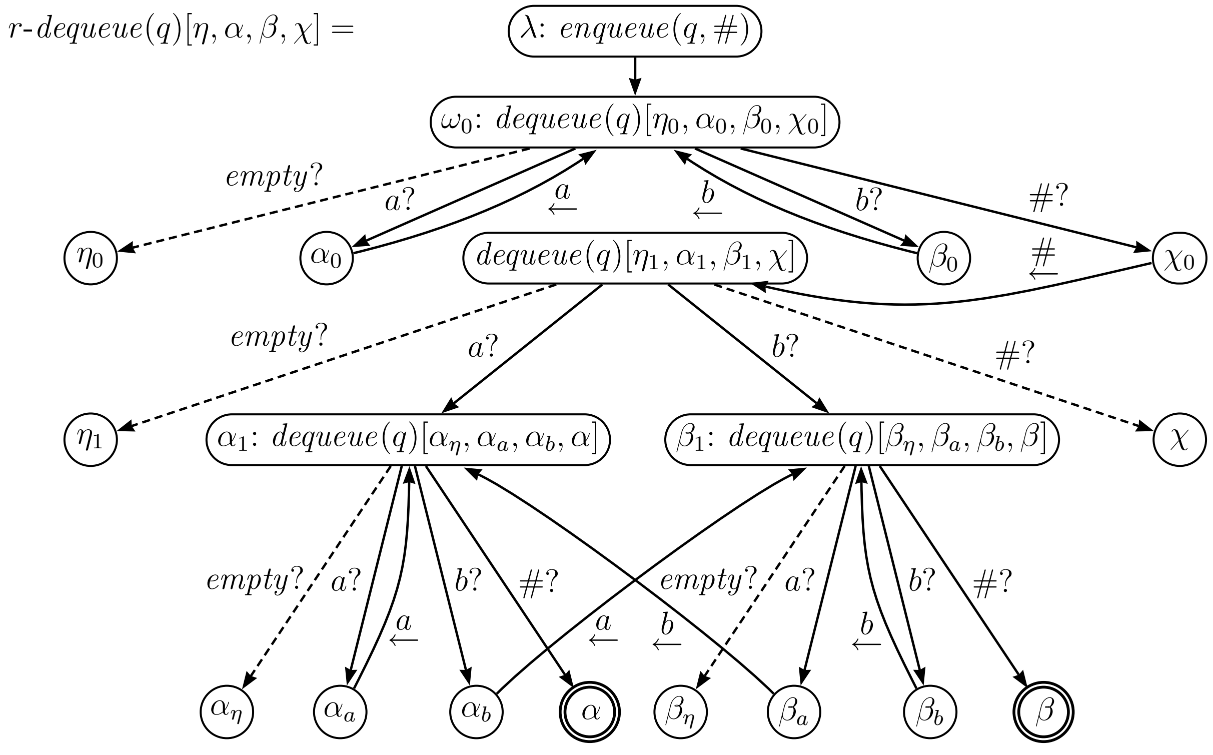

r-dequeue(q)[η, α, β, χ] =
λ: enqueue(q, #)
ω0: dequeue(q)[η0, α0, β0, χ0]
empty?
a?
a
b
b?
#?
#
η0
α0
dequeue(q)[η1, α1, β1, χ]
β0
χ0
empty?
a?
b?
#?
η1
α1: dequeue(q)[αη, αa, αb, α]
β1: dequeue(q)[βη, βa, βb, β]
χ
empty?
a?
a
b?
#?
a
b
empty?
a?
b
b?
#?
αη
αa
αb
α
βη
βa
βb
β

Fig. 18. Diagram that illustrates the statements for the *r-dequeue*$(q)[\eta, \alpha, \beta, \chi]$ operation. We have written $\overset{a}{\leftarrow}$, $\overset{b}{\leftarrow}$, and $\overset{\#}{\leftarrow}$, instead of *enqueue*(q, a), *enqueue*(q, b), and *enqueue*$(q, \#)$, respectively. The dashed arcs (those with label *empty*?) are not followed because the queue q is never empty. There is no need for statements with label η or η_0 or η_1 or α_η or β_η or χ.

$r\,s\,r \;\rightarrow\; r\,r\,r\,r$ where $\vartheta = \{S_1/\varepsilon,\ S_2/r\}$,
$r\,s\,r\,s \;\rightarrow\; r\,r$ where $\vartheta = \{S_1/s\,r,\ S_2/\varepsilon\}$,
$r\,s\,r\,s \;\rightarrow\; r\,s\,r\,r\,s\,r$ where $\vartheta = \{S_1/\varepsilon,\ S_2/r\,s\}$, and
$s\,s\,r$ cannot be rewritten because it does not match $r\,S_1\,s\,S_2$.

We may also define, as usual, the reflexive and transitive closure of $\rightarrow$, denoted by $\rightarrow^*$.

Now we give the formal definition of a Post System. A Post System G consists of:

- a finite alphabet Σ,

- a finite set T of auxiliary symbols,

- a finite subset A of Σ^*, called the set of *axioms*, and

- a finite set of *productions* of the form (†), where the fixed strings are from the set V^*, where $V = \Sigma \cup T$.

The splitting of the alphabet V used for the fixed strings, into the sets Σ and T, will be clarified in an example below. Indeed, we will see that the auxiliary symbols in T may be used as separators between substrings of Σ^*. They play the role of separators between the adjacent cells of a Turing Machine tape, while the symbols in Σ are used for the contents of the cells. We say that a word w in Σ^* is generated by the Post System G iff there exists an axiom a_1 in the set A of axioms in G such that $a_1 \rightarrow^* w$, using the productions of G.

A Post System for producing the strings in Σ^* with $\Sigma = \{a, b\}$, which are palindromes, is as follows. $T = \emptyset$. The axioms are: ε, a, b. The productions are: $S_1 \rightarrow a\, S_1\, a$, and $S_1 \rightarrow b\, S_1\, b$.

A Post System for producing the valid addition equalities, that is, strings of the form: $1^m + 1^n = 1^{m+n}$, is as follows.

$\Sigma = \{1, +, =\}$. $T = \emptyset$. The axiom is $1+1 = 11$. The productions are:

$S_1{+}S_2 = S_3 \quad\rightarrow\quad 1\, S_1{+}S_2 = 1\, S_3$

$S_1{+}S_2 = S_3 \quad\rightarrow\quad S_1{+}1\, S_2 = 1\, S_3$

For instance, we get: $1{+}1 = 11 \quad\rightarrow\quad 11{+}1 = 111 \quad\rightarrow\quad 11{+}11 = 1111$.

A Post System for producing binary representations of positive natural numbers, given in unary notation, is as follows.

$\Sigma = \{1, 0, =\}$. $T = \{¢, \$, \#\}$. The axioms are the unary representations of the positive natural numbers, that is, A is $\{1, 11, 111, \ldots\}$. The productions are as follows.

The following production $P1$ is for making two copies of the input number S_1.

$P1.\ \ S_1 \rightarrow S_1\, ¢\, \$\, S_1\, \#$

The following production $P2$ is for dividing by 2 the number between \$ and #, and placing the result between ¢ and \$.

$P2.\ \ S_1\, ¢\, S_3\, \$\, 1\, 1\, S_4\, \#\, S_2 \rightarrow S_1\, ¢\, S_3\, 1\, \$\, S_4\, \#\, S_2$

The following productions $P3$ and $P4$ are for placing 0 or 1 in the desired binary representation, depending on the fact that the given number is even (that is, no 1 is left between \$ and #) or odd (that is, one 1 is left between \$ and #). The substring between \$ and # is then initialized using the quotient S_3 of the previous division, for performing a subsequent division by 2.

$P3.\ \ S_1\, ¢\, S_3\, \$\, \#\, S_2 \rightarrow S_1\, ¢\, \$\, S_3\, \#\, 0\, S_2$

$P4.\ \ S_1\, ¢\, S_3\, \$\, 1\, \#\, S_2 \rightarrow S_1\, ¢\, \$\, S_3\, \#\, 1\, S_2$

The following production $P5$ is for writing the result of the computation as desired.

$P5.\ \ S_1\, ¢\, \$\, \#\, S_2 \rightarrow S_1 = S_2$.

For instance, we have:

111111 → (by $P1$) 111111¢\$111111# → (by $P2$) 111111¢1\$1111#
→ (by $P2$) 111111¢11\$11# → (by $P2$) 111111¢111\$#
→ (by $P3$) 111111¢\$111#0 → (by $P2$) 111111¢1\$1#0
→ (by $P4$) 111111¢\$1#10 → (by $P4$) 111111¢\$#110
→ (by $P5$) 111111 = 110.

From the above sequence of rewritings, we see that the auxiliary symbols ¢, \$, and # of our Post System are used as delimiters of four substrings: (i) the substring to the left of ¢, (ii) the substring to the right of ¢ and to the left of \$, (iii) the substring to the right of \$ and to the left of #, and (iv) the substring to the right of #.

We may encode the input n and the output $f(n)$ of a Turing computable function f from N to N, as the two sequences: 1^n and $1^{f(n)}$, separated by the symbol &. Using this encoding, we may say that the function f from N to N is *Post computable* iff there exists a Post System with alphabet $\Sigma = \{1, \&\}$ such that from the axiom & we can derive the string: $1^n \& 1^{f(n)}$ for any $n \in N$. We have that every Turing computable function from N to N is Post computable.

Vice versa, by encoding sequences of symbols of a given finite alphabet Σ into natural numbers, we have that every Post computable function is a Turing computable function.

Theorem 13. [Post Normal Form Theorem] There is not loss of computational power if one restricts the productions of a Post System to be of the form:

$g\, S \;\rightarrow\; S\, h$

where the fixed strings g and h are in $(\Sigma \cup T)^*$. Thus, one variable symbol S is sufficient.

Post Systems whose productions are of the form:

$$S_1\, g\, S_2 \;\rightarrow\; S_1\, h\, S_2 \qquad (\dagger\dagger)$$

provide an alternative presentation of Chomsky's formal grammars. Indeed, the set of terminal and nonterminal symbols of a given grammar is the set Σ of the corresponding Post System, and a given grammar production: $g \;\rightarrow\; h$ is simulated by the Post production ($\dagger\dagger$).

9.3 Unlimited Register Machines

An *Unlimited Register Machine*, URM for short, can be viewed as a collection of an infinite number of registers, say $R_1, \ldots, R_n, \ldots$, each holding a natural number. The content of the register R_i, for $i \geq 1$, will be denoted by r_i. A URM is associated with a *finite* sequence of *instructions* $P = \langle I_1, \ldots, I_p \rangle$, with $p \geq 1$, each of which is of the form:

(i) $Z(n)$, which makes r_n to be 0, or

(ii) $S(n)$, which increases r_n by 1, or

(iii) $\mathit{Assign}(m, n)$, which makes r_m to become equal to r_n (that is, $r_m := r_n$), or

(iv) $\mathit{Jump}(m, n, j)$, which does not change the content of any register, and if $r_m = r_n$ then the next instruction to be executed is I_j, with $1 \leq j \leq p$, otherwise, the next instruction to be executed is the one which follows the instruction $\mathit{Jump}(m, n, j)$ in the given sequence P.

The finite sequence of instructions is also called a *program*. A URM works by executing instructions, as usual, according to the given program P. The first instruction to be executed is I_1.

It is possible to define, in the usual way, the computation of a URM for a given program and an initial content of the registers. The computation of a URM stops iff there is no next instruction to be executed.

An URM M computes the function f from N^k to N which is defined as follows: $f(a_1, \ldots, a_k) = b$ iff starting from the values $a_1, \ldots, a_k$ in the registers $R_1, \ldots, R_k$, respectively, and the value 0 in all other registers, the computation of M eventually stops and we get b as the content of the register R_1.

One can show that the set of functions computed by URM's does not change if we do not allow *Assign* instructions.

An important result about URM's is that the set of functions from from N^k to N computed by URM's, is the set of the Turing computable functions from N^k to N.

9.4 Markov Systems

A *Markov System* can be viewed as a particular kind of type 0 grammar. The idea of a Markov System is to make deterministic the nondeterministic process of replacing a substring by another substring when constructing a word generated by a type 0 grammar, starting from the axiom of the grammar.

A Markov System is a quintuple $\langle \Sigma, V_N, R, R_1, R_2 \rangle$, where Σ is the set of terminal symbols, and V_N is the set of nonterminal symbols. Let V be the set $V_N \cup \Sigma$. R is the *finite sequence* $r_1, \ldots, r_n$ of rules, each of which is a pairs of words in $V^* \times V^*$. The sequence R is assumed to be the concatenation of the two disjoint subsequences R_1 and R_2.

We define a rewriting relation $\rightarrow$ on $V^* \times V^*$ as follows: $u \rightarrow v$ iff we get the string v from the string u by rewriting (as usually done with grammar productions) a substring of u according to the following two constraints:
(i) we have to use the *leftmost* rule in R (that is, the rule with smallest subscript) among those which can be applied, and
(ii) we have to apply that rule at the *leftmost* occurrence (if it can be applied in more than one substring of u).

We also define a *restricted transitive closure* of $\rightarrow$, denoted by $\rightarrow^{[*]}$, as the usual transitive closure with the condition that the last rewriting step, and that step only, is performed by using a rule in R_2.

A Markov System computes the function f from Σ^* to Σ^* which is defined as follows:

$f(u) = v$ iff $u \rightarrow^{[*]} v$.

One can show that the set of functions from Σ^* to Σ^* computed by the Markov Systems is the set of the Turing computable functions from Σ^* to Σ^*.

10 Church Thesis

In Sections 7 and 8 we have shown that there exists a bijection: (i) between Turing Machines and Type 0 grammars and (ii) between Turing Machines and Random Access Machines, which preserves the computed functions. Actually, as we already mentioned, there exists a bijection also between Turing Machines and each of the models of computation we have introduced in the previous section, that is: (i) Post Machines [25], (ii) Post Systems [6, page 61], (iii) Unlimited Register Machines, (iv) Markov Systems (see [6, page 65] and [24, page 263]), so that the set of computable functions from N to N is preserved.

Also Partial Recursive Functions which we will introduce in the following chapter, and other models of computations such as: (i) Lambda Calculus [5] and (ii) Combinatory Logic [5], define the same set of computable functions from N to N.

The fact that all these models of computations which have been introduced in the literature for capturing the informal notion of 'effective computability', rely on quite different intuitive ideas, and yet there exists a bijection between any two of them, and that bijection preserves the computed functions, is a strong evidence of the validity of the following statement, called *Church Thesis* or *Turing Thesis*:

Church Thesis. *Every function which, in intuitive terms, is 'effectively computable' is also Turing computable.*

Note that it is not possible to formally prove the Church thesis (and this is why it is called a thesis, rather than a theorem) because the notion of being 'effectively computable' is not given in a formal way.

Chapter 2

Partial Recursive Functions

11 Preliminary Definitions

Let D be a (not necessarily proper) subset of N^k for some $k \geq 0$.
A *partial function* f of arity k (that is, of k arguments) from N^k to N is a subset of $D \times N$, with $D \subseteq N^k$, such that for all x in D there exists a unique m in N such that $\langle x, m\rangle \in f$. For every $x \in D$ we say that f is *convergent in* x, or f is *defined in* x, or $f(x)$ is *convergent*, or $f(x)$ is *defined.*

The *domain* of a partial function f which is a subset of $D \times N$, such that for all x in D there exists a unique m in N such that $\langle x, m\rangle \in f$, is D. We also say that D is the subset of N^k where f is *defined.*

We say that a partial function f is *divergent in* x, or f is *undefined in* $x,$ or $f(x)$ is *divergent*, or $f(x)$ is *undefined*, or $f(x) =$ undefined, iff x does not belong to the domain of f.

A partial function of arity k is said to be *total* iff its domain is N^k.

The *codomain* of a partial function from N^k to N is N. The *range* of a partial function f from N^k to N is the set $\{m \,|\, \langle x, m\rangle \in f\}$. The element m of the range of f such that $\langle x, m\rangle \in f$, is also called the *value of* f *in* x. Instead of $\langle x, m\rangle \in f$, we also write $m = f(x)$.

Each partial function of arity 0, which is total, is identified with an element of N. In formal terms, for each $m \in N$, we identify the function $\{\langle\langle\rangle, m\rangle\}$ with m. The domain of each partial function f of arity 0 from N^0 to N which is a total function, is N^0, that is, the singleton $\{\langle\rangle\}$.

There exists a unique partial function of arity 0 from N^0 to N which is *not* total. It is the function which is undefined in the argument $\langle\rangle$. Unless otherwise specified, when we will say 'function' we will mean 'partial function'.

We write the application of the function f of arity k to the argument x as $f(x)$ or fx. If x is the k-tuple $\langle x_1, \ldots, x_k\rangle$ we also write $f(x)$ as $f(x_1, \ldots, x_k)$ or $fx_1 \ldots x_k$ or $f(z, x_k)$, where z is the $(k-1)$-tuple $\langle x_1, \ldots, x_{k-1}\rangle$.

We sometimes use extra parentheses, and we write an expression e as (e).

Below we will define the set PRF_k of the so called *partial recursive functions* of arity k, for any $k \geq 0$. It is a subset of the set of all partial functions of arity k.

The set PRF of all partial recursive functions is:

$\bigcup_{k\geq 0} \mathrm{PRF}_k$

By a total partial recursive function we mean a function in PRF which is total. Often in the literature a total partial recursive function is also called a total recursive function, or simply, a recursive function.

There exists a bijection, call it τ, between N^* and N (see below). Thus, there is a bijection between the set PRF of all partial recursive functions of zero or more arguments and the set PRF_1 of all partial recursive functions of one argument (that is, from N to N).

We will see below that: (i) this bijection τ and its inverse τ^{-1} are total partial recursive functions (actually, they are primitive recursive functions (see Definition 1 on page 57)), and (ii) the class of the partial recursive functions is closed under composition.

In what follows we will consider also a class C of properties of partial recursive functions. In order to study such class C of properties, without loss of generality, we will restrict ourselves to partial recursive functions of arity 1, because for that class C we have that every partial recursive function $g(n)$ from N to N has a property P in C iff the partial recursive function, say $\widetilde{g}(x) = g(\tau(x))$ from N^k to N, with $k \geq 0$ and $\tau(x) = n$, has property P.

For denoting functions we will often use the so called *lambda notation*. This means that a function which given the argument n, returns the value of the expression e, is written as the *lambda term* $\lambda n.e$. In order to make it explicit that the argument n may occur in the expression e, we will also write $\lambda n.e[n]$, instead of $\lambda n.e$. We will allow ourselves to say the function e, instead of the function $\lambda n.e$, when the argument n is understood from the context.

Let us now introduce some elementary notions concerning the lambda notation which will be useful below.

Given a set K of constants with arity, with k ranging over K, and a set V of variables, with v ranging over V, a *lambda term* e is either

a constant k or

a variable v or

an abstraction $\lambda v.e$ or

an application $(e_1 e_2)$, also denoted $e_1(e_2)$ or simply $e_1 e_2$.

For every lambda terms p and e, we define the notion of '*p occurs in e*' (or '*p is a subterm of e*' or '*e contains p*') as follows:

(i) p occurs in p, (ii) if p occurs in e_1 or e_2 then p occurs in $(e_1 e_2)$, and (iii) if p occurs in e or p is the variable v then p occurs in $\lambda v.e$.

An occurrence of a variable v in a term e is said to be *bound* iff it occurs in a subterm of e of the form $\lambda v.e_1$. An occurrence of a variable v in a term e is said to be *free* iff it is not bound. For instance, the leftmost occurrence of v in $v\,(\lambda v.(vx))$ is free, while the other two occurrences are bound.

Given a lambda term e the set of its *free variables*, denoted $FV(e)$, can be inductively defined as follows:

(constant) $FV(k) = \emptyset$,
(variable) $FV(v) = \{v\}$,
(abstraction) $FV(\lambda v.e) = FV(e) - \{v\}$,
(application) $FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$.

The result of the application of the function $\lambda n.e[n]$ to the argument m is $e[m/n]$, where by $e[m/n]$ we mean the expression e where all *free* occurrences of n in e have been replaced by m. Thus, for instance, (i) $(\lambda n.n+1)5 = 5+1$, and (ii) $(\lambda n.(n+((\lambda n.n+1)3)))\,5 = (5+((\lambda n.n+1)3)) = 5+(3+1) = 9$.

Note that the name of a bound variable is insignificant, that is, $\lambda x.e$ is the same as $\lambda y.e[y/x]$. Thus, for instance, $(\lambda n.(n+((\lambda n.n+1)3))) = (\lambda n.(n+((\lambda x.x+1)3)))$.

Functions of zero arguments returning the value of the expression e are written as $\lambda().e$ or simply e. For $r \geq 2$ we will write $\lambda x_1 \ldots x_r.e$ or $\lambda x_1, \ldots, x_r.e$ as an abbreviation for the lambda term $\lambda x_1.(\ldots(\lambda x_r.e)\ldots)$.

12 Primitive Recursive Functions

Definition 1. [Primitive Recursive Function] The set of the *primitive recursive functions* is the smallest set, call it PR, of functions, which includes:

(i) the *zero* functions z_k of k arguments, that is, $\lambda x_1 \ldots x_k.0$, for all $k \geq 0$,

(ii) the *successor* function s of one argument, that is, $\lambda x.x+1$, and

(iii) the i-th *projection* functions π_{ki} of k arguments, that is, $\lambda x_1 \ldots x_k.x_i$, for all $k \geq 1$ and all i such that $1 \leq i \leq k$, and it is closed under *composition* and *primitive recursion*, that is,

(iv) (*composition*) if the function h of $k \geq 1$ arguments is in PR and the k functions $g_1, \ldots, g_k$, each of them being of $m\,(\geq 0)$ arguments, are in PR, then also the following function f of m arguments

$$f(x_1, \ldots, x_m) = h(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$$

is in PR, and

(v) (*primitive recursion*) if a function h of $k+1$ arguments and a function g of $k-1$ arguments are in PR for some $k \geq 1$, then also the function f of k arguments such that:

$$f(0, x_2, \ldots, x_k) = g(x_2, \ldots, x_k)$$
$$f(x_1+1, x_2, \ldots, x_k) = h(x_1, x_2, \ldots, x_k, f(x_1, x_2, \ldots, x_k))$$

is in PR.

Obviously, we have that each function in PR is a function from N^k to N for some $k \geq 0$. The *identity function* $\lambda x.x$ from N to N is primitive recursive, because it is the projection function π_{11}.

Lemma 1. All partial functions of arity k, for $k \geq 0$, which are constant functions, are primitive recursive, that is, for all $n \geq 0$ and $k \geq 0$, $\lambda x_1 \ldots x_k.n$ is primitive recursive.

Proof. By induction on the number k of arguments.
(Basis) For $k = 0$ we have that: (i) $\lambda().0$ is primitive recursive by Definition 1 (i), and (ii) for all $n \geq 0$, $(\lambda().n+1)() = s((\lambda().n)())$, where for any 0-ary function f we denote by $f()$ the application of f to the empty tuple of arguments. Thus, by Definition 1 (iv) we have that for all $n \geq 0$ if the function $\lambda().n$ is primitive recursive then also the function $\lambda().n+1$ is primitive recursive.

(Step) Let us consider the constant function $f_m = \lambda x_1 \ldots x_m.n$. Then the function $f_{m+1} = \lambda x_1 \ldots x_m x_{m+1}.n$ is defined as follows:

$f_{m+1}(0, x_2, \ldots, x_{m+1}) = f_m(x_2, \ldots, x_{m+1})$

$f_{m+1}(x_1+1, x_2, \ldots, x_{m+1}) = \pi_{m+2\,m+2}(x_1, x_2, \ldots, x_{m+1}, f_{m+1}(x_1, x_2, \ldots, x_{m+1}))$. □

Lemma 2. Primitive recursive functions are closed under: (i) addition of dummy arguments, (ii) substitution of constants for variables, (iii) identification of arguments, and (iv) permutation of arguments.

Proof. The reader may convince himself that the statements of this lemma hold by looking at the following examples.
(i) Addition of dummy arguments. For instance, $\lambda xy.f(x,y)$ is obtained from $\lambda x.g(x)$ where for all x and y, $f(x,y) = g(x)$, by composing primitive functions because $f(x,y) = g(\pi_{21}(x,y))$.
(ii) Substitution of constants for variables. For instance, $\lambda x.g(x)$ is obtained from $\lambda xy.f(x,y)$ where for all x, $g(x) = f(x,3)$, by composing primitive functions because $g(x) = f(\pi_{11}(x), (\lambda x.3)(x))$.
(iii) Identification of arguments. For instance, $\lambda x.g(x)$ is obtained from $\lambda xy.f(x,y)$ where for all x, $g(x) = f(x,x)$, by composing primitive functions because $g(x) = f(\pi_{11}(x), \pi_{11}(x))$.
(iv) Permutation of arguments. For instance, $\lambda xy.g(x,y)$ is obtained from $\lambda xy.f(x,y)$ where for all x and y, $g(x,y) = f(y,x)$, by composing primitive functions because $g(x,y) = f(\pi_{22}(x,y), \pi_{21}(x,y))$. □

When showing that a function is primitive recursive we will allow ourselves to make use of Lemma 2 and to adopt abbreviated notations as we now indicate.

For instance, we will show that the addition function is primitive recursive by writing the following equations:

$sum(0,y) = y$

$sum(x{+}1,y) = sum(x,y) + 1$ (by primitive recursion)

instead of

$sum(0,y) = \pi_{22}(0,y)$

$sum(x+1,y) = sp(x,y,sum(x,y))$ (by primitive recursion)

where: (i) $\lambda y.\pi_{22}(0,y)$ is primitive recursive because it is obtained from $\pi_{22}(x,y)$ by replacing the variable x by the constant 0 (see Lemma 2 above), and (ii) $sp(x,y,z)$ is primitive recursive because

$sp(x,y,z) = (\lambda x.x{+}1)(\pi_{33}(x,y,z))$ (by composition)

If we apply Lemma 2 the Composition Schema (Definition 1.iv) may be generalized by allowing some of the g_i's to be functions of some, but *not all* the variables $x_1, \ldots, x_m$. Analogously, the Primitive Recursion Schema (Definition 1.v) may be generalized by allowing the function h to have as arguments only some of the $k{+}1$ arguments $x_1, x_2, \ldots, x_k$, $f(x_1, x_2, \ldots, x_k)$.

The functions defined by equations like the ones of Case (v) of Definition 1, which are called *recursive equations*, are assumed to be evaluated in the *call-by-value* mode, also called *inside-out and left-to-right* mode. For more details the reader may look at [33].

This means that the evaluation of an expression proceeds by:
(i) considering the *leftmost* subexpression, say u, among all *innermost* subexpressions which match a left-hand-side of a recursive equation, and
(ii) replacing u by $(R\vartheta)$, if u is equal to $(L\vartheta)$ for some recursive equation $L = R$ and substitution ϑ.

Actually, since all primitive recursive functions are total, the mode of evaluation of the recursive equations does not matter. However, it does matter in the case of the partial recursive functions (see Section 13) because of the presence of the minimalization operator which may lead to non-terminating computations.

It can be shown that: (i) the functions defined by composition and primitive recursion are uniquely determined by their recursive equations (this result is stated in the so called *Recursion Theorem* which can be found, for instance, in [9]), and (ii) for all x in the domain of f we can compute by using the call-by-value mode of evaluation the unique element m of N such that $f(x) = m$.

It follows from Definition 1 that the set of primitive recursive functions is denumerable. Indeed, for the set PR we have the following domain equation [32, Chapter 4]:

$$\mathrm{PR} = N \ + \ \{1\} \ + \ N\times N + \mathrm{PR}\times\mathrm{PR}^* + \mathrm{PR}\times\mathrm{PR}$$

The five summands of the right hand side of this equation correspond to the cases (i)–(v), respectively, of Definition 1 on page 57. It can be shown by induction that every primitive recursive function is total.

Almost all ‘algorithmic functions’ of ordinary mathematics are primitive recursive. Here is the proof for some of them. The proof is given by exhibiting the recursive equations of those functions. We leave it to the reader to fill in the details and to show how these recursive equations can be replaced by suitable applications of the composition and primitive recursion schemata as indicated above for the function *sum*. We have the following equations.

1. (addition)

$$sum(0,y) = y \qquad sum(x{+}1,y) = sum(x,y) + 1 \qquad \text{(by primitive recursion)}$$

We will also write $sum(x,y)$ as $x{+}y$. Note that we have overloaded the symbol $+$, because $x{+}1$ also denotes the unary successor function $\lambda x.s(x)$.

2. (predecessor: $x \neg 1 = \mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ x{-}1$)

$$0 \neg 1 = 0 \qquad (x{+}1) \neg 1 = x \qquad \text{(by primitive recursion)}$$

It is a primitive recursive function because:

$$0 \neg 1 = (\lambda().0)() \text{ and } (x{+}1) \neg 1 = \pi_{21}(x, x{\neg}1).$$

3. (proper subtraction: $x \neg y = \mathit{if}\ y \geq x\ \mathit{then}\ 0\ \mathit{else}\ x{-}y$)

$$x \neg 0 = x \qquad x \neg (y{+}1) = (x \neg y){\neg}1 \qquad \text{(by primitive recursion)}$$

As often done in the literature we have overloaded the symbol $\neg$ because we have used it for proper subtraction and predecessor. If $x \geq y$ we will feel free to write $x - y$, instead of $x{\neg}y$. Indeed, if $x \geq y$ then $x{\neg}y = x - y$, where $-$ is the usual subtraction operation between natural numbers we learnt at school.

4. (multiplication)

$0 \times y = 0 \qquad (x+1) \times y = (x \times y) + y$ (by primitive recursion)

We will also write $x \times y$ as $x\,y$.

5. (exponentiation)

$x^0 = 1 \qquad x^{y+1} = x^y x$ (by primitive recursion)

6. (factorial)

$0! = 1 \qquad (x+1)! = (x+1) \times (x!)$ (by primitive recursion)

7. (sign: $sg(x) = \mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ 1$)

$sg(0) = 0 \qquad sg(x+1) = 1$ (by primitive recursion)

8. (negated sign: $nsg(x) = \mathit{if}\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ 0$)

$nsg(0) = 1 \qquad nsg(x+1) = 0$ (by primitive recursion)

9. (minimum between two natural numbers: min) (by composition)

$min(x, y) = x \neg (x \neg y)$

10. (maximum between two natural numbers: max) (by composition)

$max(x, y) = x + (y \neg x)$

11. (absolute difference: $|x - y| = max(x, y) \neg min(x, y)$) (by composition)

$|x - y| = (x \neg y) + (y \neg x)$

We have that: $|x - y| = |y - x|$.

The following properties relate the functions sg, nsg, and the *if-then-else* construct. For any $a, b \in N$, we have that:

$$\mathit{if}\ x{=}0\ \mathit{then}\ a\ \mathit{else}\ b\ =\ a\ nsg(x) + b\ sg(x)$$
$$\mathit{if}\ x{=}y\ \mathit{then}\ a\ \mathit{else}\ b\ =\ a\ nsg(|x - y|) + b\ sg(|x - y|)$$

In particular, for any $a \in N$,

$$\mathit{if}\ x{=}y\ \mathit{then}\ 0\ \mathit{else}\ a\ =\ \mathit{if}\ x{\neq}y\ \mathit{then}\ a\ \mathit{else}\ 0\ =\ a\ sg(|x - y|)$$
$$\mathit{if}\ x{=}y\ \mathit{then}\ a\ \mathit{else}\ 0\ =\ \mathit{if}\ x{\neq}y\ \mathit{then}\ 0\ \mathit{else}\ a\ =\ a\ nsg(|x - y|)$$

12. ($rem(x, y) =$ integer remainder when x is divided by y)

We assume that for any $x \in N$, $rem(x, 0) = x$. Thus, rem is a total function from N^2 to N. We have that: for any $y \in N$,

$$\begin{aligned} rem(0, y) &= 0 \\ rem(x + 1, y) &= \mathit{if}\ (rem(x, y)+1) \neq y\ \mathit{then}\ rem(x, y)+1\ \mathit{else}\ 0 \end{aligned}$$

Thus,

$$\begin{aligned} rem(0, y) &= 0 \\ rem(x + 1, y) &= (rem(x, y)+1)\ sg(|(rem(x, y)+1) - y|) \end{aligned}$$

This last equation can be rewritten as follows:

$rem(x+1, y) = g(rem(x, y), y)$ where

$g(y, z) = (y+1)\ sg(|(y+1) - z|)$

and the function g is primitive recursive because it can be obtained by composition.

Note that $rem(x,y) < y$ iff $y > 0$. In particular, $rem(x,y) < y$ does *not* hold if $y = 0$. Indeed, for any $x \in N$, we have that $rem(x,0) = x$ and $x \not< 0$.

13. ($quot(x,y)$ = integer quotient when x is divided by y)

We assume that for any $x \in N$, $quot(x,0) = x$. Thus, $quot$ is a total function from N^2 to N. We have that:

$$\begin{aligned} quot(0,y) &= 0 \\ quot(x+1,y) &= \mathit{if}\ \ (rem(x,y)+1 \neq y)\ \ \mathit{then}\ \ quot(x,y)\ \ \mathit{else}\ \ quot(x,y)+1 \\ &= quot(x,y)\ +\ (\mathit{if}\ \ (rem(x,y)+) \neq y)\ \ \mathit{then}\ 0\ \ \mathit{else}\ 1) \end{aligned}$$

Note that, analogously to what happens in standard arithmetics, for all $x, y \in N$, we have that:

$$x = y\ \times\ quot(x,y)\ +\ rem(x,y).$$

14. (divisibility*:* $div(x,y) = \mathit{if}\ \ x$ is divided by $y\ \ \mathit{then}\ 1\ \ \mathit{else}\ 0$)

We assume that x is divided by 0 iff $x = 0$, that is, $div(x,0) = 1$ iff $x = 0$. Thus, div is a total function from N^2 to N. We have that:

$$div(x,y) = nsg(rem(x,y))$$

Here are some examples of applications of the functions *rem*, *quot*, and *div*.

(i) For the division $0/4$ we have: $rem(0,4) = 0$ and $quot(0,4) = 0$.
(ii) For the division $4/0$ we have: $rem(4,0) = 4$ and $quot(4,0) = 0$.
(iii) $div(0,x) = nsg(rem(0,x))$= 1 for any $x \geq 0$.
(iv) $div(x,0) = nsg(rem(x,0)) = 0$ for $x \neq 0$.

Definition 2. [Predicate and Characteristic Function] A *predicate* P on N^k for $k \geq 0$, is a subset of N^k. If $x \in P$ we also write $P(x)$ and we say that $P(x)$ holds or $P(x)$ is true. Given a predicate P on N^k, its *characteristic function*, denoted f_P, is a function of arity k from N^k to N, defined as follows:

$$f_P(x) = \mathit{if}\ \ x \in P\ \ \mathit{then}\ 1\ \ \mathit{else}\ 0, \quad \text{for any } x \in N^k.$$

Definition 3. [Primitive Recursive Predicate] A predicate on N^k for $k \geq 0$, is said to be *primitive recursive* iff its characteristic function is primitive recursive.

For instance, the predicates $x < y$, $eq(x,y)$ (also written as $x = y$), and $x > y$, for $x, y \in N$ are primitive recursive. Indeed, $f_{x<y} = sg(y \neg x)$, $f_{eq(x,y)} = nsg(|x – y|)$, and $f_{x>y} = sg(x \neg y)$.

Lemma 2 can easily be extended to primitive recursive predicates. We leave that extension to the reader.

Lemma 3. The set of the primitive recursive predicates is closed under negation, (finite) disjunction, and (finite) conjunction.

Proof. (i) $f_{not\,P}(x) = 1 \neg f_P(x)$, (ii) $f_{P\,or\,Q}(x) = sg(f_P(x) + f_Q(x))$, and
(iii) $f_{P\,and\,Q}(x) = f_P(x) \times f_Q(x)$. □

We have that the predicate $x \leq y$ is primitive recursive, because $x \leq y$ iff ($x < y$ or $eq(x,y)$).

Let us introduce the following definition. In this definition and in what follows, for any pair a and b of natural numbers, with $a \leq b$, $[a,b]$ denotes the subset $\{a, a+1, \ldots, b\}$ of N.

Definition 4. [Bounded Existential and Universal Quantification] Given a predicate P on N^{k+1} for $k \geq 0$, the *bounded existential quantification* of P is the predicate Q on N^{k+1} such that $Q(x,n)$ iff $\exists i \in [0,n]\, P(x,i)$.

Analogously, for the *bounded universal quantification* by replacing $\exists$ by $\forall$.

Lemma 4. Primitive recursive predicates are closed under bounded existential quantification and bounded universal quantification.

Proof. If $f_P(x,n)$ is the characteristic function of the predicate $P(x,n)$ then the characteristic function $f_Q(x,n)$ of the predicate $Q(x,n) = \exists i \in [0,n]\, P(x,i)$ is defined by primitive recursion as follows:

$$f_Q(x,0) = f_P(x,0) \qquad f_Q(x,n{+}1) = f_Q(x,n) + f_P(x,n{+}1).$$

For the bounded universal quantification, it is enough to recall that Lemma 3 and the fact that $\forall i \in [0,n]\, P(x,i)$ is the same as *not*$(\exists i \in [0,n]\;$ *not*$(P(x,i)))$. □

Lemma 5. [Definition by cases] Let us consider the set $\{\langle g_i(x), P_i(x)\rangle \mid 1 \leq i \leq r$ and $1 \leq r\}$, where for $1 \leq i \leq r$, $g_i(x)$ is a primitive recursive function from N^k to N and $P_i(x)$ is a primitive recursive predicate denoting a subset of N^k. Let us assume that for each x in N^k one and only one of the predicates $P_i(x)$, for $1 \leq i \leq r$, holds. Then the function f which satisfies the following equations:

$$\begin{aligned} f(x) &= g_1(x) \quad \textit{if} \;\; P_1(x)\\ &= g_2(x) \quad \textit{if} \;\; P_2(x)\\ &\;\;\ldots\\ &= g_r(x) \quad \textit{if} \;\; P_r(x) \end{aligned}$$

is primitive recursive.

Proof. The case where $r = 1$ is obvious, because the definition of $f(x)$ corresponds to the equation $f(x) = g_1(x)$ *if* true, that is, $f(x) = g_1(x)$. For $r > 1$ we have the following. Let $c_i(x)$ be the characteristic function of $P_i(x)$ for $1 \leq i \leq r$. Then

$$f(x) = (\ldots(g_1(x)c_1(x) + g_2(x)c_2(x)) + \ldots + g_r(x)c_r(x)).$$ □

Note that the case where $r = 2$ corresponds to the equation:

$$f(x) = \textit{if } P(x) \textit{ then } g_1(x) \textit{ else } g_2(x)$$

which we also write as:

$$\begin{aligned} f(x) &= g_1(x) \quad \textit{if}\, P(x)\\ &= g_2(x) \quad \textit{otherwise.} \end{aligned}$$

Remark 1. In a definition by cases of a primitive recursive function, the evaluation of the predicates $P_1(x)$, $P_2(x)$, …, and $P_r(x)$ is done sequentially from left to right. Since the corresponding characteristic functions are total, the order of evaluation of these predicates is *not* important. However, if we allow predicates with characteristic functions which are partial functions, as we will do in the sequel, this order of evaluation is important, because otherwise, the definition by cases may fail to specify a partial function. □

Now we define a new operator, called the *bounded minimalization operator*.

Definition 5. **[Bounded Minimalization]** Given a predicate Q, subset of N^{k+1}, the *bounded minimalization operator*, denoted *min*, allows us to define a function f of arity $k{+}1$, with $k \geq 0$, such that: for any $x \in N^k$ and $n \in N$,

$$\begin{aligned} f(x,n) &= \text{the minimum } y \in [0,n] \text{ such that } Q(x,y) \text{ holds} \\ &\qquad\quad \textit{if there exists } y \in [0,n] \text{ such that } Q(x,y) \text{ holds} \\ &= 0 \quad \textit{otherwise} \end{aligned}$$

The function $f(x,n)$ is denoted by $\textit{min}\ y \in [0,n]\, Q(x,y)$.

In the *otherwise* clause we could have chosen for $f(x,n)$ a value different from 0 without modifying the theory of the partial recursive functions. The choice of 0 is convenient when formulating Lemma 9 and, in particular, when establishing Lemma 9 Point (iv).

Lemma 6. If the predicate $Q(x,y)$ subset of N^{k+1}, for $k \geq 0$, is primitive recursive then the function $f(x,n) = \textit{min}\ y \in [0,n]\ Q(x,y)$ is primitive recursive.

Proof. Let us consider the function

$$\begin{aligned} g(x,n,m) &= m & & \textit{if } \exists i \in [0,n]\ Q(x,i) \\ &= n{+}1 & & \textit{if not}(\exists i \in [0,n]\ Q(x,i)) \textit{ and } \ Q(x,n{+}1) \\ &= 0 & & \textit{if not}(\exists i \in [0,n{+}1]\ Q(x,i)) \end{aligned}$$

The function g is primitive recursive because it is defined by cases starting from primitive recursive functions and primitive recursive predicates. We will show that $f(x,n)$ is primitive recursive by showing that it is equal to the following function h defined by primitive recursion, as follows:

$$\begin{aligned} h(x,0) &= 0 \\ h(x,n{+}1) &= g(x,n,h(x,n)) \end{aligned}$$

(Recall that g is primitive recursive). By unfolding the definition of g, we have that the function h is also defined by the equations:

$$\begin{aligned} h(x,0) &= 0 \\ h(x,n{+}1) &= h(x,n) & & \textit{if } \exists i \in [0,n]\ Q(x,i) \\ &= n{+}1 & & \textit{if not}(\exists i \in [0,n]\ Q(x,i)) \textit{ and } Q(x,n{+}1) \\ &= 0 & & \textit{if not}(\exists i \in [0,n{+}1]\ Q(x,i)) \end{aligned}$$

Now, recalling the definition of the bounded minimalization operator *min*, we also have that:

$$\begin{aligned} f(x,0) &= \textit{min}\, y \in [0,0]\, Q(x,y) = \textit{if } \ Q(x,0) \ \textit{then } 0 \ \textit{else } 0 \\ f(x,n{+}1) &= \textit{if } \exists i \in [0,n]\, Q(x,i) \ \textit{then } f(x,n) \ \textit{else} \\ & \ \textit{if } Q(x,n{+}1) \ \textit{then } n{+}1 \ \textit{else } 0 \end{aligned}$$

Thus, we conclude that $h(x,n) = f(x,n)$ for all $x \in N^k$, and $n \in N$, because they are defined by the same recursion equations. □

By using Lemma 6 we can show that many other mathematical functions are primitive recursive. For instance, we have that the $\textit{integersquareroot}(x)$, defined as the largest number whose square does not exceed x, is a primitive recursive function. Indeed,

$$\textit{integersquareroot}(x) = \textit{min}\ y \in [0,x]\ (y^2 \leq x) \ \text{ and } \ ((y{+}1)^2 > x).$$

Lemma 7. Let $h(n)$ be a primitive recursive function from N to N and $f(x,m)$ a primitive recursive function from N^{k+1} to N of the form: $min\ y \in [0,m]\ Q(x,y)$, where x is a k-tuple of numbers and m is a number. Then the function

$$g(x,n) = min\ y \in [0,h(n)]\ Q(x,y)$$

is primitive recursive.

Proof. The function $g(x,n)$ is defined by composition as follows:

$$g(x,n) = f(x,h(n)) = f(\pi_{k+1\,1}(x,n), \ldots, \pi_{k+1\,k}(x,n),\ h(\pi_{k+1\,k+1}(x,n))). \qquad \square$$

Definition 6. [**Bounded Sum and Bounded Product**] Let f be a primitive recursive function from N^{k+1} to N. Let x denote a k-tuple of numbers, and n and r denote two numbers. The *bounded sum of the function* $f(x,n)$ *below* r, denoted $\Sigma_{n<r}f(x,n)$, is defined as follows:

$$\Sigma_{n<r}f(x,n) = \textit{if } r\!=\!0 \textit{ then } 0 \textit{ else } f(x,0) + f(x,1) + \ldots + f(x,r\!-\!1)).$$

The *bounded product of the function* $f(x,n)$ *below* r, denoted $\Pi_{n<r}f(x,n)$, is defined as follows:

$$\Pi_{n<r}f(x,n) = \textit{if } r\!=\!0 \textit{ then } 1 \textit{ else } f(x,0) \times f(x,1) \times \ldots \times f(x,r\!-\!1)).$$

Often the qualification *'below r'* is dropped and we will simply say '*the bounded sum of* $f(x,n)$' or '*the bounded product of* $f(x,n)$'.

Lemma 8. Given a primitive recursive function f from N^{k+1} to N and a number r, the bounded sum $\Sigma_{n<r}f(x,n)$ and the bounded product $\Pi_{n<r}f(x,n)$ are primitive recursive functions. The same holds if in the summations and in the products we replace $<$ by $\leq$.

Proof. We have the following primitive recursive equations:

$$\Sigma_{n<0}f(x,n) = 0 \quad \Sigma_{n<(r+1)}f(x,n) = \Sigma_{n<r}f(x,n) + f(x,r), \quad \text{and}$$
$$\Pi_{n<0}f(x,n) = 1 \quad \Pi_{n<(r+1)}f(x,n) = \Pi_{n<r}f(x,n) \times f(x,r).$$

For instance, if we write $\Sigma_{n<r}f(x,n)$ as $S(x,r)$, we have that:

$$S(x,0) = 0$$
$$S(x,r\!+\!1) = S(x,r) + f(x,r).$$

$S(x,r)$ is a primitive recursive function, because $S(x,r) + f(x,r) = sum1(x,r,S(x,r))$ and $sum1(x,r,m)$ is a primitive recursive function. Indeed, we have that:

$$sum1(x,r,m) = sum(\pi_{33}(x,r,m), f(\pi_{31}(x,r,m), \pi_{32}(x,r,m))).$$

When replacing $<$ by $\leq$ we have:

$$\Sigma_{n\leq r}f(x,n) = \Sigma_{n<(r+1)}f(x,n), \text{ and } \Pi_{n\leq r}f(x,n) = \Pi_{n<(r+1)}f(x,n).$$

Thus, we have that for all $r \geq 0$, the functions $\lambda x.\Sigma_{n\leq r}f(x,n)$ and $\lambda x.\Pi_{n\leq r}f(x,n)$ are primitive recursive by Lemma 7. □

Lemma 9. The following functions are primitive recursive:

(i) $D(n)$ = the number of divisors of n, according to the following definition of divisor:
y is a divisor of x iff $div(x,y) = 1$ and $y \leq x$.
Thus, we have: $D(0) = 1$, $D(1) = 1$, $D(2) = 2$, $D(3) = 2$, $D(4) = 3$, $D(5) = 2, \ldots$

(ii) $Pr(n) = \textit{if } (n\!\geq\!2 \text{ and } n \text{ is prime}) \textit{ then } 1 \textit{ else } 0$

(iii) $p(n)$ (also written as p_n) = the n-th prime number, with $n\!\geq\!0$, defined as follows:
$p_0 = 0$, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, $p_4 = 7$, $p_5 = 11, \ldots$

(iv) $e(x,n) = \mathit{if}\ n{=}0\ \mathit{or}\ x{=}0\ \mathit{then}\ 0\ \mathit{else}$ the exponent of p_n in the prime factorization of x.
Thus, for instance, $e(x,1)$, $e(x,2)$, and $e(x,3)$ are the exponents of 2, 3, and 5, respectively, in the prime factorization of x.
(v) The inverse functions of the so called *pair function* $C(x,y) = 2^x(2y+1)\neg 1$. Note that $C(x,y)$ is a bijection from N^2 to N.

Proof. (i) $D(x) = D1(x,x)$ where the function $D1$ is defined as follows:
$D1(x,0) = \mathit{div}(x,0)$ and $D1(x,n{+}1) = D1(x,n) + \mathit{div}(x,n{+}1)$.
(ii) $Pr(x) = \mathit{if}\ D(x) = 2\ \mathit{then}\ 1\ \mathit{else}\ 0$.
$D(x){=}2$ means that $x{>}1$ and the only divisors of x are 1 and x. Thus,
$Pr(x) = \mathit{nsg}(|D(x){-}2|)$.
(iii) $p_0 = 0$ and $p_{n+1} = \mathit{min}\, z \in [0, p_n! + 1]\ (z > p_n \text{ and } Pr(z))$.
This is a definition by primitive recursion, because: (a) ($z > p_n$ and $Pr(z)$) is a primitive recursive predicate, (b) we may apply Lemma 7 for $h(n) = p_n!+1$, and (c) the remainders of the n divisions of $p_n! + 1$ by $p_0, p_1, \ldots$, and p_n, respectively, are all different from 0, and thus, in the interval $[p_n{+}1,\ p_n!{+}1]$ there is at least one prime.
(iv) $e(x,n) = \mathit{min}\, z \in [0,x]\ (\mathit{div}(x, p_n^{z+1}) = 0)$.
(v) Let n be $C(x,y)$. Thus, $n{+}1$ is $2^x(2y{+}1)$. We have that: $x = e(n{+}1, 1)$, because $2y{+}1$ is odd, and $y = \mathit{quot}(\mathit{quot}(n{+}1, 2^x)\neg 1, 2)$. □

An element $\langle x_1, x_2, \ldots, x_k\rangle$ of N^k, for $k \geq 0$, can be encoded by the following element of N:
$\beta(x_1, x_2, \ldots, x_k) = \mathit{if}\ k{=}0\ \mathit{then}\ 1\ \mathit{else}\ (p_1^{x_1+1} \times p_2^{x_2+1} \times \ldots \times p_k^{x_k+1})$.
From the value of $\beta(x_1, x_2, \ldots, x_k)$, call it B, we can get the values of k and those of the x_i's for $i = 1, \ldots, k$, as follows:
$k = \mathit{if}\ B{=}1\ \mathit{then}\ 0\ \mathit{else}\ \mathit{min}\, z \in [0,B]\ (e(B, z{+}1) = 0)$, and
for $i = 1, \ldots, k$, $x_i = e(B,i)\neg 1$.
The encoding β from N^* to N is *not* a bijection. Indeed, for instance, no element x exists in N^* such that $\beta(x) = 3$. The same holds for $\beta(x) = 0$.

Lemma 10. Given an element $\langle x_1, x_2, \ldots, x_k\rangle$ of N^k, for any $k \geq 0$, let τ be the following function with arity k:

$$\tau(x_1, x_2, \ldots, x_k) = \mathit{if}\ k{=}0\ \mathit{then}\ 0\ \mathit{else}\ 2^{x_1} + 2^{x_1+x_2+1} + 2^{x_1+x_2+x_3+2} + \ldots + 2^{x_1+x_2+\ldots+x_k+k\neg 1}.$$

The function τ is a bijection from N^* to N and realizes an encoding of k-tuples of natural numbers into natural numbers. The function τ and its inverse τ^{-1} are primitive recursive functions.

Proof. If $k = 0$ then $\tau(x_1, x_2, \ldots, x_k) = \tau(\langle\rangle) = 0$. If $k > 0$ then $\tau(x_1, x_2, \ldots, x_k) > 0$. It immediate to see that τ is primitive recursive.

Given the value of $\tau(x_1, x_2, \ldots, x_k)$, we can compute τ^{-1}, that is, we can derive the values of $x_1, x_2, \ldots$, and x_k, by using primitive recursive functions only as we now indicate.

We have that $\tau^{-1}(0) = \langle\rangle$.
Given any $x \geq 0$, we have that x has a unique expression of the form:
$x = \alpha(0,x)\, 2^0 + \ldots + \alpha(i,x)\, 2^i + \ldots$

where, for any $i \geq 0$, $\alpha(i, x) \in \{0, 1\}$. Thus, for $i \geq 0$, the values of $\alpha(i, x)$ give us the binary expansion of the number x. In particular, for all $j \geq i$ such that $x < 2^i$, we have that $\alpha(j, x) = 0$.

Given any $x > 0$, it has a unique expression of the form:

$$x = 2^{\,b(1,x)} + 2^{\,b(2,x)} + \ldots + 2^{\,b(M(x),x)}$$

with $0 \leq b(1, x) \leq b(2, x) \leq \ldots \leq b(M(x), x)$ and $1 \leq M(x)$, where $M(x)$ is the number of the powers of 2 occurring in the binary expansion of x. The number x has also a unique expression of the form:

$$x = 2^{\,a(1,x)} \;+\; 2^{\,a(1,x)+a(2,x)+1} \;+\; 2^{\,a(1,x)+a(2,x)+a(3,x)+2} + \ldots$$
$$+\, 2^{\,a(1,x)+a(2,x)+\ldots+a(M(x),x)+k \neg 1}$$

where $1 \leq M(x)$ and for $j = 1, \ldots, M(x)$, we have that $a(j, x) \geq 0$.

For example, if $x = 2^5 + 2^8 + 2^{15}$ $(= 33056)$ we have that:

$b(1, x) = 5, \quad b(2, x) = 8, \quad b(3, x) = 15,$
$M(x) = 3,$
$a(1, x) = 5, \quad a(2, x) = 2, \quad a(3, x) = 6,$
$k = 3,$ and
$\tau^{-1}(x) = \langle 5, 2, 6 \rangle.$

It is easy to check that $\tau(\tau^{-1}(x)) = x$. Indeed, $\tau(\langle 5, 2, 6 \rangle) = 2^5 + 2^8 + 2^{15}$.

Now we show that the functions $b(i, x)$, $M(x)$, and $a(j, x)$ are primitive recursive and this will imply that the function τ^{-1} is primitive recursive.

(i) $\alpha(0, x) = \mathit{rem}(x, 2) \qquad \alpha(i{+}1, x) = \alpha(i, \mathit{quot}(x, 2))$ for $i \geq 0$

(ii) Let us denote $b(M(x), x)$ by $E(x)$. We have that, for $x > 0$, $E(x)$ is the highest exponent of 2 occurring in the binary expansion of x. We have:

$$E(x) = \mathit{if}\ x{=}0\ \mathit{then}\ 0\ \mathit{else}\ \mathit{min}\ z \in [0, x]\ (2^{z+1} > x).$$

(iii) $M(x) = \mathit{if}\ x{=}0\ \mathit{then}\ 0\ \mathit{else}\ K(E(x), x)$

where $K(0, x) = \alpha(0, x)$ and $K(j{+}1, x) = K(j, x) + \alpha(j{+}1, x)$ for any $j \geq 0$.

For $x > 0$, $K(E(x), x)$ is the number of the powers of 2 occurring in the binary expansion of x. We also have that:

$$M(x) = \Sigma_{j<x}\, \alpha(j, x).$$

(Recall that since for any $x \geq 0$, $x < 2^x$ we have that: for any $i \geq x$, $\alpha(i, x) = 0$).

(iv) $b(i, x) = \mathit{if}\ x{=}0\ \mathit{or}\ i{=}0\ \mathit{or}\ i{>}M(x)\ \mathit{then}\ 0\ \mathit{else}$
$\qquad\qquad \mathit{if}\ i{=}1\ \mathit{then}\ r\ \mathit{else}\ b(i{-}1, x{-}2^r)$

where $r = \mathit{min}\, z \in [0, x]\ (\alpha(z, x) > 0)$. Thus, r is the exponent of the smallest power of 2 occurring in the binary expansion of x. Note that if the condition '$x{=}0$ *or* $i{=}0$ *or* $i{>}M(x)$' is false then i is in the interval $[1, M(x)]$. For i in $[1, M(x)]$, we also have:

$$b(i, x) = \mathit{min}\, z \in [0, x]\ ((\Sigma_{k \leq z}\, \alpha(k, x)) = i).$$

(v) The function $a(j, x)$ can be expressed as follows:

$a(j, x) = \mathit{if}\ x{=}0\ \mathit{or}\ j{=}0\ \mathit{or}\ j{>}M(x)\ \mathit{then}\ 0\ \mathit{else}$
$\qquad\qquad \mathit{if}\ j{=}1\ \mathit{then}\ b(1, x)\ \mathit{else}\ ((b(j, x) \neg b(j{-}1, x)) \neg 1).$ □

Example 1. In order to better understand some of the expressions we have defined in the proof of the above Lemma 10 on page 65, let us consider the number 13. We have that:

$13 = 2^0 + 2^2 + 2^3$

$13 = 2^{b(1,13)} + 2^{b(2,13)} + 2^{b(3,13)}$

We also have that:

$b(1,13) = 0$	$b(2,13) = 2$	$b(3,13) = 3$ (see below)	
$\alpha(0,13) = 1$	$\alpha(1,13) = 0$	$\alpha(2,13) = 1$	$\alpha(3,13) = 1$
for $z = 0$:	for $z = 1$:	for $z = 2$:	for $z = 3$:
$\sum_{i=0}^{z} \alpha(i,13) = 1$	$\sum_{i=0}^{z} \alpha(i,13) = 1$	$\sum_{i=0}^{z} \alpha(i,13) = 2$	$\sum_{i=0}^{z} \alpha(i,13) = 3$

$E(13) = 3$

$M(13) = 3$

Here is the computation of the value of $b(3,13)$.

First, we take the binary expansion of 13, which is 1011 (the most significant bit is to the right). The leftmost bit 1 is in position 0. We have: $2^0 = 1$ and $13-1 = 12$. We get: $b(3,13) = b(2,12)$.

Then, in order to compute $b(2,12)$, we take the binary expansion of 12 which is 0011. The leftmost bit 1 is in position 2. We have: $2^2 = 4$ and $12 - 4 = 8$. We get: $b(2,12) = b(1,8)$.

In order to compute $b(1,8)$, we take the binary expansion of 8 which is 0001. The leftmost bit 1 is in position 3. We have: $2^3 = 8$ and $8-8 = 0$. We get: $b(1,8) = 3$.

Therefore, $b(3,13) = b(2,12) = b(1,8) = 3$. □

Exercise 1. (i) Show that primitive recursive functions are closed under mutual recursive definitions. Without loss of generality, let us consider the following functions g and h with one argument only:

$$\begin{aligned} g(0) &= a \\ g(n{+}1) &= r(n, g(n), h(n)) \\ h(0) &= b \\ h(n{+}1) &= s(n, g(n), h(n)) \end{aligned}$$

where the functions r and s are primitive recursive.
We have that the function $f(n) = 2^{g(n)}\, 3^{h(n)}$ is primitive recursive, because:

$$\begin{aligned} f(0) &= 2^a 3^b \\ f(n{+}1) &= 2^{g(n+1)} 3^{h(n+1)} = \\ &= 2^{r(n,g(n),h(n))} 3^{s(n,g(n),h(n))} = \\ &= 2^{r(n,u,v)} 3^{s(n,u,v)} \text{ where } \langle u, v\rangle = \langle e(f(n),1),\ e(f(n),2)\rangle \end{aligned}$$

where the definition of the function $\lambda x, n.\, e(x,n)$ is given in Lemma 9 Point (iv). Thus, $g(n) = e(f(n),1)$ (because $g(n)$ is the exponent of 2 in the prime factorization of $f(n)$) and $h(n) = e(f(n),2)$ (because $h(n)$ is the exponent of 3 in the prime factorization of $f(n)$). □

Exercise 2. Show that the Fibonacci function is primitive recursive. We have that:

$$\begin{aligned} \mathit{fib}(0) &= 0 \\ \mathit{fib}(1) &= 1 \\ \mathit{fib}(n{+}2) &= \mathit{fib}(n{+}1) + \mathit{fib}(n) \end{aligned}$$

We have that the function $f(n) = 2^{\mathit{fib}(n+1)}\,3^{\mathit{fib}(n)}$ is primitive recursive, because:

$$\begin{aligned} f(0) &= 2^1 3^0 \\ f(n+1) &= 2^{\mathit{fib}(n+2)} 3^{\mathit{fib}(n+1)} = \\ &= 2^{u+v} 3^u \text{ where } \langle u, v\rangle = \langle \mathit{fib}(n+1),\, \mathit{fib}(n)\rangle \\ &= 2^{u+v} 3^u \text{ where } \langle u, v\rangle = \langle e(f(n), 1),\, e(f(n), 2)\rangle \end{aligned}$$

where the definition of the function $\lambda x, n.\, e(x, n)$ is given in Lemma 9 Point (iv).

Thus, $\mathit{fib}(n) = e(f(n), 2)$. □

Remark 2. The *course-of-value recursion* can be reduced to primitive recursion (see the result by R. Péter presented in [22, page 273]), in the sense that we get an equivalent definition of the set of the primitive recursive functions if in Definition 1 we replace the second equation of (v) by the following one:

$$\begin{aligned} &f(x_1{+}1, x_2, \ldots, x_k) = \\ &\quad h(x_1, x_2, \ldots, x_k, f(x_1, x_2, \ldots, x_k), f(x_1{-}1, x_2, \ldots, x_k), \ldots, f(0, x_2, \ldots, x_k)). \end{aligned}$$ □

13 Partial Recursive Functions

Let us begin by introducing the *minimalization operator* μ with respect to a class C of functions.

Definition 7. [Minimalization μ] Given a function t from N^{k+1} to N for some $k \geq 0$, in the class C of functions, the *minimalization operator* μ allows us to define a new function f from N^k to N, denoted $\lambda x.(\mu y.[t(x, y) = 0])$, such that for any $x \in N^k$,

$$\begin{aligned} f(x) &= \text{the minimum } y \text{ such that } t(x,y){=}0 & &\textit{if} \text{ there exists } y \in N \text{ such that } t(x,y){=}0 \\ &= \text{undefined} & &\textit{otherwise} \end{aligned}$$

In the literature there exists a different minimalization operator with respect to a class C of functions. It is called μ^*.

Definition 8. [Minimalization μ^*] Given a function t from N^{k+1} to N for some $k \geq 0$, in the class C of functions, the *minimalization operator* μ^* allows us to define a new function f^* from N^k to N, denoted $\lambda x.(\mu^* y.[t(x, y) = 0])$, such that for any $x \in N^k$,

$$\begin{aligned} f^*(x) &= \text{the minimum } y \text{ such that (i) for all } d{\leq}y, t(x,d) \text{ is defined and (ii) } t(x,y){=}0, \\ &\qquad \textit{if} \text{ there exists } y \in N \text{ such that} \\ &\qquad\quad \text{(ii.1) for all } d \leq y,\ t(x,d) \text{ is defined and} \\ &\qquad\quad \text{(ii.2) } t(x,y) = 0 \\ &= \text{undefined} \qquad \textit{otherwise} \end{aligned}$$

Remark 3. When applying the operators μ and μ^* we need to know the class C of functions to which the function t belongs. □

Remark 4. When introducing the minimalization operators, one could consider, instead of the predicate $\lambda x, y.[t(x, y) = 0]$, the predicate $\lambda x, y.[t(x, y) = r]$, for some fixed number $r \in N$. Any choice of a natural number r, different from 0, does not modify the theory of the partial recursive functions. □

Now we define the set PRF of the partial recursive functions and, for any $k \geq 0$, the sets PRF_k of the partial recursive functions of arity k.

Definition 9. [**Partial Recursive Function**. **Version 1**] The set PRF of all *partial recursive functions* (*p.r.f.*'s, for short) is the smallest set of partial functions, subset of $\bigcup_{k\geq 0}(N^k \rightarrow N)$, which includes: (i.1) the zero functions, (i.2) the successor function, and (i.3) the projection functions (as defined in Definition 1) and (ii) it is closed with respect to the following operations: (ii.1) composition, (ii.2) primitive recursion, and (ii.3) minimalization μ with respect to the subset of the partial recursive functions themselves which are *total* functions.

The set PRF_k of the partial recursive functions of arity k, for any $k \geq 0$, is the subset of PRF which includes all partial recursive functions of arity k, that is, from N^k to N. □

It follows from the above Definition 9 that the set of the partial recursive functions is denumerable. Indeed, for the set PRF we have the following domain equation [32, Chapter 4]:

$$\mathrm{PRF} = N \ + \ \{1\} \ + \ N^* \times N + \mathrm{PRF} \times \mathrm{PRF}^* + \mathrm{PRF} \times \mathrm{PRF} + \mathrm{TPRF}$$

where TPRF is the subset of PRF which includes all partial recursive functions which are total. Obviously, the cardinality of TPRF is not smaller than that of N (because for all $n \in N$, the constant function $\lambda x.n$ is in TPRF) and it is not larger than that of PRF.

One can show that the set PRF_k of partial recursive functions is *not* closed under μ operations with respect to the class PRF_{k+1} of functions. (Note that PRF_{k+1} includes also partial functions.) In particular, for $k=1$ there exists a partial recursive function f in PRF_2 such that $\lambda i.(\mu y.[f(i,y) = 0])$ is *not* a partial recursive function. One such p.r.f. f is defined as follows:

$$\lambda i \lambda y. \mathit{if}\ y=1 \lor (y=0 \land g_i(i) \text{ is defined}) \ \mathit{then}\ 0 \ \mathit{else} \text{ undefined} \qquad (\dagger)$$

where g_i is the i-th partial recursive function [33, page 42].

Remark 5. We can speak of the i-th p.r.f. g_i, for any $i \geq 0$, because the set of the p.r.f.'s is recursively enumerable (see Definition 1 on page 87), that is, there is Turing Machine M such for all natural number n, M returns a p.r.f., and for every p.r.f. g there exists a natural number n such that M returns g. (Note that M may return the same g for two or more different input values of n.)

The fact that the set of the p.r.f.'s is recursively enumerable is a consequence of the following two points: (Point 1) each of the five cases considered in the definition of the primitive recursive functions (see Definition 1 on page 57), refers to either a particular function (case (ii)) or a denumerable number of functions (cases (i), (iii), (iv), and (v)), and (Point 2) the minimalization operator depends on a total p.r.f. and the set of the total p.r.f.'s is denumerable.

In the above expression (†) the word 'undefined' stands for '$\mu y.[1(y) = 0]$', where $\lambda y.1$ is the total function of one argument returning the value 1, for any $y \in N$. Moreover, as we will see in the proof of Theorem 3 on page 71, by using (a variant of) the Kleene T-predicate (see page 70) and the minimalization operator μ, we can replace '$g_i(i)$ is defined' by a predicate whose characteristic function is a p.r.f. Thus, we have a syntactic way of establishing that the expression (†) actually defines a p.r.f.

An alternative way of establishing that the expression (†) defines a p.r.f. is by applying the bijection between the set of the p.r.f.'s and the set of the Turing Machines (see Section 17 on page 80). That bijection tells us that there exists an algorithm (that is, a Turing Machine) that computes the natural number m, if any, such that $m = g_i(i)$ and that algorithm terminates iff m exists.

The following equivalent definition of the set PRF of the partial recursive functions is equivalent to Definition 9.

Definition 10. [Partial Recursive Function. Version 2] The set PRF of the partial recursive functions is the smallest set of partial functions, subset of $\bigcup_{k\geq 0}(N^k \rightarrow N)$, which: (i) includes all primitive recursive functions, and (ii) it is closed with respect to: (ii.1) composition, (ii.2) primitive recursion, and (ii.3) minimalization μ^* with respect to the set PRF itself of the partial recursive functions. □

In the above Definition 10, instead of saying 'includes all primitive recursive functions', we could have said, as in Definition 9, 'includes the zero functions, the successor function, and the projection functions'.

Theorem 1. Each partial recursive function f of one argument can be expressed as:
$\lambda n.p(\mu y.[t(n,y) = 0])$
for some *primitive recursive functions* p and t (possibly depending on f).

As a consequence of Theorem 1, we have an alternative proof that the set of p.r.f.'s is denumerable. Indeed, the partial recursive functions are as many as the pairs of primitive recursive functions, that is, $|N \times N|$ (which is equal to $|N|$).

Remark 6. A statement similar to the one of Theorem 1 where the function p is omitted, does *not* hold. In particular, there exists a p.r.f. f such that no total p.r.f. g exists such that f is $\lambda n.(\mu y.[g(n,y) = 0])$ [33, page 37]. □

Remark 7. Given a partial recursive function f, Theorem 1 tells us how to compute the value of $f(n)$ for any $n \in N$. Indeed, let f be equal to $\lambda n.p(\mu y.[t(n,y) = 0])$. Then, in order to compute $f(n)$ we compute the sequence of values: $t(n,0), t(n,1), \ldots$, until we find the smallest y, call it $\overline{y}$, such that $t(n,\overline{y}) = 0$. Then $f(n) = p(\overline{y})$. Obviously, if such a value of y does *not* exist, then $f(n)$ is undefined. □

Theorem 2. [Kleene Normal Form Theorem] The set PRF_1 of the p.r.f.'s of arity 1 can be obtained by choosing two fixed primitive recursive functions P (of arity 1) and T (of arity 3). We have that $\mathrm{PRF}_1 = \{\lambda n.P(\mu y.[T(i,n,y) = 0]) \mid i \geq 0\}$. □

The function T of Theorem 2 is (a variant of) the so called *Kleene T-predicate*. T is a total function satisfying the following property: $T(i,n,y) = 0$ iff i is the encoding of a Turing Machine M such that there exists $m \in N$ which is computed by M with input n in *fewer* than $q(y)$ computation steps, where $\lambda y.q(y)$ is fixed primitive recursive function, independent of i, n, and y [33, page 30]. In a sense the function T incorporates in its definition also the definition of the function q. (Note also that Turing Machines can be encoded by natural numbers.) Since the function T is primitive recursive, Theorem 2 holds also for μ^*, instead of μ.

For all $i \geq 0$ the expression $\lambda n.P(\mu y.[T(i,n,y)=0])$ defines the i-th partial recursive function of arity 1. Every partial recursive function of arity 1 has infinitely many indices, that is, for every p.r.f. f from N to N there exists an infinite subset I of N such that for all $i \in I$ we have that $f = \lambda n.P(\mu y.[T(i,n,y)=0])$.

Some partial recursive functions are total. Indeed, this is the case when, with reference to Definition 7 on page 68 and Definition 9 on page 69, for all $x \in N^k$ there exists a minimum value of y such that $t(x,y)=0$.

A total p.r.f. is also called a *recursive function.* In the literature, unfortunately, partial recursive functions sometimes are called *recursive functions* tout court. The reader should be aware of this ambiguous terminology.

The set of the total partial recursive functions can be defined as in Definition 9 with the extra condition that the total partial recursive function t which is used in the minimalization operator μ, satisfies the following property:

$\forall x \in N.\, \exists y \in N.\, t(x,y)=0.$

Definition 11. [Recursive Predicate] A predicate, subset of N^k, is said to be *recursive* iff its characteristic function from N^k to N is a recursive function.

There are p.r.f.'s which are always undefined, that is, p.r.f.'s whose domain is empty. One such function is, for instance, $\lambda n.(\mu y.[t(n,y)=0])$ where $t = \lambda n,y.1$ (that is, t is the constant function of two arguments which returns 1).

There exists a total p.r.f. which is *not* a primitive recursive function. A p.r.f. which is total and is not primitive recursive is the Ackermann function *ack* from $N \times N$ to N defined as follows:

for any $m,n \in N$,

$$\begin{aligned} &ack(0,n) &&= n+1\\ &ack(m+1,0) &&= ack(m,1)\\ &ack(m+1,n+1) &&= ack(m,ack(m+1,n)) \end{aligned}$$

We omit the proof that the Ackermann function is a p.r.f. which is *not* a primitive recursive function. The proof that the Ackermann function is not primitive recursive is based on the fact that it grows faster than any primitive recursive function (see also [6, page 194]), that is, for all primitive recursive function p there exists n such that $ack(n,n) > p(n)$.

The total p.r.f.'s are as many as $|N|$.

Theorem 3. There exists a p.r.f. f from N to N which cannot be extended to a total p.r.f., call it g, from N to N, in the sense that for all $n \in N$ if $f(n)$ is convergent then $f(n) = g(n)$.

Proof. Assume that for each p.r.f. f_i there exists a total p.r.f., call it f_{i^*}, which extends f_i. Let us consider the function h such that for all $i \geq 0$,

$h(i) =_{def}$ *if* $f_i(i)$ is defined *then* $f_i(i)+1$ *else* undefined

where f_i is the i-th p.r.f. from N to N. Thus, we have that:

$h(i) =_{def}$ *if* $\mu y.[T(i,i,y)=0] \geq 0$ *then* $f_i(i)+1$ *else* $\mu y.[1(y)=0]$

where $\lambda y.1$ is the constant function of one argument which returns 1. From this expression and recalling that the Kleene T-predicate is total p.r.f., we have that the function h is a p.r.f. Let us assume that the function h is the p.r.f. of index r, that is, $h = f_r$. By assumption, we have that there exists the p.r.f. f_{r^*} which is total. Since (a) if $f_i(i)$ is defined then h is defined and (b) f_{r^*} extends h, we have that for all $i \geq 0$,

if $f_i(i)$ is defined *then* ($f_{r^*}(i)$ is defined and $f_{r^*}(i) = f_i(i)+1$).

If we consider $i = r^*$ we get that:

if $f_{r^*}(r^*)$ is defined *then* ($f_{r^*}(r^*)$ is defined and $f_{r^*}(r^*) = f_{r^*}(r^*)+1$),

which is a contradiction because f_{r^*} is a total p.r.f. and, thus, $f_{r^*}(r^*)$ is defined. □

From Definition 9 it follows that given a partial recursive function g of $k+1$ arguments which is total, then the function $\lambda x_1, \ldots, x_k.(\mu y.[g(x_1, \ldots, x_k, y) = 0])$ is a partial recursive function which is not necessarily total because it may be the case that for all $x_1, \ldots, x_k$, no y exists such that $g(x_1, \ldots, x_k, y) = 0$.

14 Partial Recursive Functions and Recursively Enumerable Sets

Definition 12. [Recursively Enumerable Sets] A set A is *recursively enumerable*, r.e. for short, iff either $A = \emptyset$ or there exists a total p.r.f. f from N to N such that $A = \mathit{range}(f)$. □

This definition motivates the name 'recursively enumerable' given to the set A. Indeed, assuming that A is not empty, we have that the elements of the recursively enumerable set A are 'enumerated, possibly with repetitions,' by a total p.r.f. f in the sense that $A = \{f(i) \mid i \in N\}$. The meaning of 'possibly with repetitions' is that in the sequence: $f(0)$, $f(1)$, $f(2), \ldots$, we may have identical elements of the set A.

Let us now introduce the notion of *dove-tailing evaluation* of a partial recursive function.

For this purpose we first need to consider a suitable *model of computation* and a notion of *computation step* relative to that model. (Actually, the basic results we will present do not depend on the particular model we choose, because they depend only on function convergence.) We assume that the notion of computation step is a primitive notion and, thus, we need not to specify it. With reference to Definition 9, we will only say that a computation step can be understood as:

- in cases (i.1)–(i.3) the evaluation of a zero function, or the successor function, or a projection function, or
- in cases (ii.1) and (ii.2) the replacement of a left-hand-side by the corresponding right-hand-side, or else
- in case (ii.3) the computation step of the function t relative to the minimalization operator μ.

Given a partial recursive function f from N to N, we say that the *bounded evaluation of* $f(i)$ for j computation steps is:

- $f(i)$ if performing *at most* j computation steps the evaluation of $f(i)$ gives us a number in N,
- the distinguished value # (not in N), otherwise.

An alternative definition of a computation step of a partial recursive function is as follows. A computation step of a p.r.f. f is a move of the Turing Machine (see page 13) that, as we will see below, is associated with the p.r.f. f.

Given a partial recursive function f from N to N, its *dove-tailing evaluation* is the construction of a two-dimensional matrix where at row $i\,(\geq 0)$ and column $j\,(\geq 0)$ we place the bounded evaluation of $f(i)$ for j computation steps (see also Figure 19 on page 73). This matrix is constructed element by element, in the increasing order of the entries which we denote by $(0), (1), (2), \ldots$, called *position numbers*. For any $i, j \in N$, at row i and column j we have the unique position number $D(i,j) = ((i+j)^2 + 3i + j)/2$.

There are two functions r and c from N to N such that for any position number $d\,(\geq 0)$ we have that d occurs in the dove-tailing matrix at row $r(d)$ and column $c(d)$. The two functions r and c are primitive recursive, because as the reader may verify, $r(d) = d - S$ and $c(d) = s - r(d)$, where:

$S = quot((s^2{+}s), 2)$ and

$s = quot(integersquareroot(8d{+}1){+}1,\ 2) - 1$

and $integersquareroot(x)$ is the natural number n such that $n^2 \leq x < (n{+}1)^2$.

computation step j:	0	1	2	3	...
argument i: 0	(0)	(1)	(3)	(6)	...
1	(2)	(4)	(7)	...	
2	(5)	(8)	...		
3	(9)	...			
...	...				

Fig. 19. The matrix of the position numbers for the dove-tailing evaluation of a p.r.f. from N to N.

Now we state an important result about the dove-tailing evaluation of any p.r.f.: there exists a partial recursive function, call it I, from $\mathrm{PRF}{\times}N \times N$ to $N{\cup}\{\#\}$ which given any partial recursive function f and the numbers i and j, computes the value of the bounded evaluation of $f(i)$ for j computation steps. The function I is a total function as we will see below.

The proof of the existence of the function I (given in [6, Chapter 5] and [34]) relies on the existence of the Universal Turing Machine and on the fact (not proved here) that there exists a bijection between the set PRF of the partial recursive functions and the set of the sequences of instructions for Turing Machines, that is, the set of the Turing Machines themselves (recall also that an instruction for a Turing Machine can be encoded by a quintuple of numbers). This bijection exists because of: (i) the injection from the set PRF to the set of the Turing Machines, (ii) the injection from the set of the Turing Machines to the set PRF, and (iii) the Bernstein Theorem [32, Chapter 1].

Note that the two injections from the set PRF to the set of the Turing Machines and vice versa, are Turing computable functions, while we do not require the bijection between from the set PRF to the set of the Turing Machines is a Turing computable function.

Since there exists a Turing computable injection from the set PRF to the set of the Turing Machines, we can associate with every p.r.f. f a unique, deterministic Turing Machine, (that is, a sequence of instructions or quintuples), which computes f (see also Section 17 on page 80).

Since a sequence of instructions can be encoded by a sequence of numbers (because an instruction is, in general, a sequence of characters and a character can be encoded by a number), and a sequence of numbers can be encoded by a single number (see the encoding function τ in Section 12 on page 65), we have that any partial recursive function, say f, can be encoded by a number, say $\tau^*(f)$, which is called the *number encoding* of f.

Thus, when we say that I takes as input a p.r.f. f we actually mean that it takes as input the number $\tau^*(f)$. The function I is a total function from $\mathrm{PRF}\times N\times N$ to $N\cup\{\#\}$. It can also be considered as a function from N to N because of the bijection between PRF and N, the bijection between N^3 and N, and the bijection between $N\cup\{\#\}$ and N.

The function I extracts from the value of $\tau^*(f)$ the sequence of instructions of the Turing Machine which computes f, and then it behaves as an *interpreter* (and this is the reason why we gave the name I to it) which uses that sequence as a program for evaluating $f(i)$ for j computation steps.

Now we give an informal proof of the existence of the function I. This proof will be based on the fact that for each partial recursive function f there exists a set of recursive equations which can be used for evaluating f (see Section 12 starting on page 57 and Remark 7 on page 70).

An informal proof of the existence of the function I. Given a p.r.f. f, it can be viewed as a set, call it Eq_f, of recursive equations constructed from the zero, successor, and projection functions, by using composition, primitive recursion, and minimalization. In particular, a function $\lambda x.g(x)$ defined by the minimalization operation of the form $\mu y.[t(x,y)=0]$ is equivalent to the following two equations:

$$g(x) = aux(x,0)$$
$$aux(x,y) = \textit{if } t(x,y)=0 \textit{ then } y \textit{ else } aux(x,y{+}1)$$

Then the set Eq_f of recursive equations can be viewed as a program for the function f. We can encode that program by a number which we also denote, by abuse of language, $\tau^*(f)$.

Let us consider the computational model related to the leftmost-innermost rewriting of subexpressions and let us consider a computation step to be a rewriting step. The initial expression e_0 (that is, the expression at computation step 0) is the number which encodes $f(i)$, for any given p.r.f. f and input number i. Then, given the number which encodes the expression e_k at the generic computation step k (≥ 0), the function I produces the number which encodes the expression e_{k+1} at the next computation step $k+1$ as follows. The function I finds in the program a recursive equation, say $L=R$, such that the leftmost-innermost subexpression u of e_k is equal to $(L\vartheta)$ for some substitution ϑ, and it replaces the subexpression u in e_k by $(R\vartheta)$.

We hope that the reader may convince himself that the operations of: (i) finding the leftmost-innermost subexpression, (ii) finding the recursive equation to be applied, and (iii) rewriting that subexpression, are all operations which can be performed starting from the number $\tau^*(f)$ and the number which encodes the expression e at hand, by the application of a suitable total partial recursive function which operates on numbers. □

Theorem 4. A set A is r.e. iff A is the domain of a p.r.f.

Proof. (*only-if part*) Let us call p the p.r.f. to be found, whose domain is A. If $A = \emptyset$ the function p from N to N is the everywhere undefined partial recursive function. This function can be denoted by the lambda term $\lambda n.(\mu y.[1(n,y) = 0])$, where for all n and y, the function $\lambda n, y.\, 1(n,y)$ returns 1. (Obviously, infinite many other lambda terms denote the everywhere undefined partial recursive function.)

If $A \neq \emptyset$ then, by definition, A is the range of a *total* p.r.f., say f. A is the domain of the partial recursive function p which is $\lambda x.\mu y.[f(y) - x = 0]$. Indeed, $x \in A$ iff $\exists y. f(y) = x$ iff p is defined in x.

(*if part*) We have that A is the domain of a p.r.f., call it g. By definition, for all $x \in A$, $g(x)$ is defined, i.e., $g(x) \in N$. We have to show that A is r.e.

If $A = \emptyset$ then there is nothing to prove because A is r.e. by definition.

If $A \neq \emptyset$ then we show that A is r.e. by constructing a total p.r.f., call it h, such that A is the range of h.

Let us first consider a list of pairs of numbers, call it G, which is constructed as follows. The initial value of G is the empty list. Then the value of G is updated by inserting in it new pairs of numbers, as we now indicate.

We first perform the dove-tailing evaluation of the function g. We will see below that we can perform the dove-tailing evaluation of the function g in an incremental way, while computing the (possibly infinite) set of pairs that constitutes the function h. Let us assume that by dove-tailing evaluation of g, we get a table whose initial, left-upper portion is depicted in Figure 20.

computation step j:	0	1	2	3	4	5	...
argument i: 0	#	#	#	#	#	#	...
1	#	#	#	#	#	...	
...	...						
x	#	#	m	m	...		
...	...						

Fig. 20. Dove-tailing evaluation of the function g.

When a value of $g(x)$, say m, different from #, is obtained in row x and column y, we insert at the right end of the list G the pair $\langle x, D(x,y)\rangle$ (here we assume that the list G grows to the right). (Recall that: $D(x,y) = ((x+y)^2 + 3x + y)/2$.) By definition of dove-tailing, if $\langle x, D(x,y)\rangle$ is in G then in G there are infinitely many pairs whose first component is x (because there are infinitely many m's in row x in Figure 20). In particular, if $m \in N$ is in row x and column y then for each column $k \geq y$, m is in row x and column k.

Since A is the domain of g, $g(x)$ is convergent for each $x \in A$, and therefore for every $x \in A$ there exists $y \in N$ (actually, there exist infinitely many y's) such that the pair $\langle x, D(x,y)\rangle$ is inserted into G.

Since A is not empty, the list G has at least one pair. Let us consider the pair of G with the smallest second component, that is, the pair which is first inserted into G during the dove-tailing evaluation of g. Let that pair be $\langle x_0, d_0\rangle$. (Recall that, by definition of the function r which computes the row of any given position number, we have that $x_0 = r(d_0)$.) Thus, the list G is of the form:

$G = [\langle x_0, d_0\rangle, \langle x_1, d_1\rangle, \ldots, \langle x_n, d_n\rangle, \ldots]$

where $\langle d_0, d_1, \ldots, d_n, \ldots\rangle$ is an infinite sequence of increasing numbers, that is, $d_0 < d_1 < \ldots < d_n < \ldots$

Note that the list G is infinite even if A is a finite set (by hypothesis, A is non-empty). We also have that for all $x \in A$ and for all $d \geq 0$ there exists $d_1 > d$ such that if $\langle x, d\rangle$ is in G then $\langle x, d_1\rangle$ is also in G.

Let us define the function h from N to N, as follows:

(1) $h(0) = x_0$

(2) $h(n+1) = \mathit{if}\ \langle x, n+1\rangle$ is in G *then* x *else* x_0

Thus, for every $n \in N$, we have that $h(n) = r(n)$ if the bounded evaluation of $g(r(n))$ returns a number within $c(n)$ computation steps, otherwise $h(n) = x_0$. As a consequence, since $A \neq \emptyset$ and the pair $\langle x_0, d_0\rangle$ occurs in G, we have that for every $n \in N$, in order to compute $h(n)$, we only need to compute a finite portion of the dove-tailing of the function g.

We have the following three facts.
(i) The function h is p.r.f. because the function I which for any $x \geq 0$ and any $n \geq 0$, performs the bounded evaluation of $g(x)$ for $c(n)$ computation steps, is a total partial recursive function from N^3 to $N \cup \{\#\}$, and so is the function which for $i = 0, \ldots, n+1$, performs the bounded evaluations of $g(r(i))$ for $c(i)$ computation steps. Thus, the predicate '$\langle x, n+1\rangle$ is in G' is recursive.
(ii) By definition of h, we have that h is a total p.r.f.
(iii) Moreover, the range of h is A. Indeed, let us consider any $x \in A$. Since A is the domain of the partial recursive function g, for that x there exists a position number d such that: (iii.1) $x = r(d)$, and (iii.2) the bounded evaluation of $g(x)$ returns an element of N (that is, it does not return #) within $c(d)$ computation steps. By (iii.2) we have that $\langle x, d\rangle$ is in G and, as a consequence, $h(d) = x$ (see Equation (2) above). □

In order to understand the ($\Leftarrow$) part of the proof of Theorem 4 and to clarify the definition of the function h, let us now consider the following example.

Example 2. Let us assume that the dove-tailing evaluation of the function g, depicted in Figure 21 below, produces the following list:

$G = [\langle 2, 8\rangle, \langle 1, 11\rangle, \langle 2, 12\rangle, \langle 1, 16\rangle, \langle 2, 17\rangle, \langle 4, 19\rangle, \ldots]$

$\langle 2, 8\rangle$ means that at position number 8, that is, at row $r(8)=2$ and column $c(8)=1$, we have computed the value of $g(2)$, which is m_8, with at most 1 computation step,
$\langle 1, 11\rangle$ means that at position number 11, that is, at row $r(11)=1$ and column $c(11)=3$, we have computed the value of $g(1)$, which is m_{11}, with at most 3 computation steps,
$\langle 2, 12\rangle$ means that at position number 12, that is, at row $r(12)=2$ and column $c(12)=2$, we have computed (again) the value of $g(2)$, which is $m_{12}=m_8$, with at most 2 computation steps,

. . .,
$\langle 4, 19\rangle$ means that at position number 19, that is, at row $r(19)=4$ and column $c(19)=1$, we have computed the value of $g(4)$, which is m_{19}, with at most 1 computation step,
. . .

computation step j:	0	1	2	3	4	5	. . .
argument i: 0	(0)	(1)	(3)	(6)	(10)	(15)	. . .
1	(2)	(4)	(7)	m_{11}	m_{16}	. . .	
2	(5)	m_8	m_{12}	m_{17}	. . .		
3	(9)	(13)	(18)	. . .			
4	(14)	m_{19}	. . .				
. . .	. . .						

Fig. 21. Dove-tailing evaluation of the function g. The r.e. set A which is the domain of g is $\{2, 1, 4, \ldots\}$. It is also the range of the function h defined by the Equations (1) and (2).

In the matrix of Figure 21 we have written, instead of the #'s, the corresponding position numbers between parentheses. The entries of that matrix different from #, which are in the same row, are all equal. Indeed, they are the value of the function g for the same input. In particular, in our matrix of Figure 21:
(i) for row 1, we have: $g(1) = m_{11} = m_{16} = \ldots$,
(ii) for row 2, we have: $g(2) = m_8 = m_{12} = m_{17} = \ldots$,
(iii) for row 4, we have: $g(4) = m_{19} = \ldots$, etc.

Since during the dove-tailing evaluation the first computed value of the function g is $g(2) = m_8$, we have that in the definition of the function h, the value of x_0 is 2. Thus,

n:	0	1	2	. . .	8	9	10	11	12	. . .	15	16	17	18	19	. . .
$h(n)$:	2	2	2	. . .	2	2	2	1	2	. . .	2	1	2	2	4	. . .

To sum up, in order to compute $h(n)$ we first perform the dove-tailing evaluation of the function g until the following two conditions are satisfied:
(i) the position at row $r(n)$ and column $c(n)$ has been filled with the value of the bounded evaluation of $g(r(n))$ with at most $c(n)$ computation steps (it may be either a number or #), and
(ii) the value of g has been computed for at least one element in A, call it x_0, (recall that A is not empty, by hypothesis) with x_0 different from #. □

Theorem 5. A set A is r.e. iff A is the range of a p.r.f.

Proof. (*only-if part*) Let us call p the p.r.f. to be found whose range is A. If $A = \emptyset$ the function p is the everywhere undefined partial recursive function. This function can be denoted by the lambda term $\lambda n.(\mu y.[1 = 0])$.

If $A \neq \emptyset$ the p.r.f which has A as range, is the total p.r.f. which exists because A is, by definition, the range of a total p.r.f.

(*if part*) We have that A is the range of a p.r.f., call it g.

If $A = \emptyset$ then A is r.e. by definition.

If $A \neq \emptyset$ then we show that A is r.e. by constructing a total p.r.f., call it h, such that A is the range of h.

The proof of this part of the theorem is like part ($\Leftarrow$) of the proof of Theorem 4, with the exception that:

(i) during the construction of the list G, when the value of $g(x)$, say m, different from #, is obtained in row x and column y, we insert at the right end of the list G the pair $\langle m, D(x,y)\rangle$, instead of the pair $\langle x, D(x,y)\rangle$, and

(ii) the function h is defined as follows:

(1′) $h(0) = m_{in}$

(2′) $h(n+1) = \textit{if } \langle m, n+1\rangle \text{ is in } G \textit{ then } m \textit{ else } m_{in}$

where m_{in} is the first component of the pair which is first inserted into G. Note that, since $A \neq \emptyset$, G must have at least one pair.

By definition, the function h is total.

We also have that A is the range of h. Indeed, for every $x \in A$, x is by hypothesis in the range of g, and thus, there exists k such that the bounded evaluation of $g(r(k))$ gives us the number x within $c(k)$ computation steps. As a consequence, for every $x \in A$ there exists $k \in N$ such that since $\langle x, k\rangle$ is in G and thus, by definition of h (see Equation 2 above), we have that for all $x \in A$ there exists $k \in N$ such that $h(k) = x$. This shows that A is the range of h. □

In order to understand the ($\Leftarrow$) part of the proof of Theorem 5, let us now consider the following example.

Example 3. Let us assume that the dove-tailing evaluation of the function g, depicted in Figure 21 above, produces the following list:

$G = [\langle m_8, 8\rangle, \langle m_{11}, 11\rangle, \langle m_8, 12\rangle, \langle m_{11}, 16\rangle, \langle m_8, 17\rangle, \langle m_{19}, 19\rangle, \ldots]$

This value of G means that during the dove-tailing evaluation
- at position 8 (and $12, 17, \ldots$) we have computed $g(2) = m_8 = m_{12} = m_{17} = \ldots$,
- at position 11 (and $16, \ldots$) we have computed $g(1) = m_{11} = m_{16} = \ldots$,
- at position 19 we have computed $g(4) = m_{19} = \ldots$, etc.

In our case we have that $m_{in} = m_8$ and $h(0) = m_8$. Thus, the values of the function h are as follows:

n:	0	1	2	...	8	9	10	11	12	...	15	16	17	18	19	...
$h(n)$:	m_8	m_8	m_8	...	m_8	m_8	m_8	m_{11}	m_8	...	m_8	m_{11}	m_8	m_8	m_{19}	...

□

Definition 13. [Recursive Set] A set $A \subseteq N$ is *recursive* iff there exists a total p.r.f., say f_A, such that $\forall x \in N.\ \textit{if } x \in A \textit{ then } f_A(x) = 1 \textit{ and if } x \notin A \textit{ then } f_A(x) = 0$.

It follows from the definition that for any given recursive set A the corresponding function f_A (which is a total p.r.f.) can be used for deciding whether or not a number is an element of A, that is, for deciding the membership problem relative to the set A.

By definition, recursive sets are the largest collection of sets for which it is possible to decide the membership problem via a total p.r.f.

Related to Definition 13 we also have the following theorem.

Theorem 6. A set A is r.e. iff there exists a partial recursive function, say f_A, such that:
$\forall x \in N.$ *if* $x \in A$ *then* $f_A(x) = 1$ *else* undefined. (†)

Proof. Let us consider the partial recursive function p which we have constructed in the proof of the *only-if part* of Theorem 4. We have that A is the domain of $p(x)$. Thus, $p(x) \in N$ iff $x \in A$, and if $x \notin A$ then $p(x)$ is undefined. Let us also consider the following function f_A:

$f_A(x) = nsg(0 \neg p(x))$.

Now we have that: (i) $f_A(x)$ is a p.r.f. because $p(x)$ is a p.r.f. and the class of the partial recursive functions is closed with respect to composition, and (ii) $f_A(x)$ satisfies (†) because by the properties of the function $p(x)$ we have that $f_A(x)=1$ iff $x \in A$, and if $x \notin A$ then f_A is undefined. (Recall that expressions are evaluated in the call-by-value mode.) □

There are sets which are r.e. and are not recursive. One such set can be constructed as follows.

Let us consider the following injective encoding, say τ^*, from the set of Pascal programs into the set N of natural numbers. Since every Pascal program is a finite sequence of characters, for each program there exists a unique corresponding finite sequence of numbers, and that sequence can be encoded by a single number (via, for instance, the above mentioned τ function).

Let us consider the set

$D = \{\tau^*(p) \mid p$ is a Pascal program and p halts when given the input $\tau^*(p)\}$.

Thus, a number n is in D iff n is the encoding of a Pascal program which halts when its input is n. One can show that D is a r.e. subset of N, but it is not a recursive subset.

Obviously, instead of Pascal programs, we may equivalently consider C++ or Java programs. Indeed, the choice of the programming language does not matter as long as we can write in it a program for any partial recursive function.

15 Recursive Enumerable Sets and Turing Computable Functions

An alternative definition of recursively enumerable sets can be given in terms of Turing Machines, instead of p.r.f.'s. Indeed, as we will indicate in Section 17, it can be shown that a function is a p.r.f. from N to N iff it can be computed by a Turing Machine, having chosen a suitable encoding of numbers by strings of tape symbols.

One can show that r.e. sets are exactly those subsets of N which can be *enumerated* by Turing Machines in the following sense: for any r.e. set $A \subseteq N$ there exists a Turing Machine T_A such that if A is the empty subset of N then for any input $n \in N$, T_A does not halt, and if A is not empty then

(i) for any $n \in N$, when given n as input, the Turing Machine T_A halts and gives us back an element of A, and

(ii) for any a if $a \in A$ there exists $n \in N$ such that when given n as input, the Turing Machine T_A halts and gives us back the element a.

Note that if a set A is r.e. there exists an infinite number of Turing Machines which enumerate A, and in particular, there exists an infinite number of Turing Machines which do not halt for all input $n \in N$.

16 Partial Recursive Functions on Strings

Instead of considering p.r.f.'s from N to N we may consider p.r.f.'s from a generic domain $M1$ to a generic domain $M2$. In particular, we may choose $M1 = M2 =$ the integers, or $M1 = M2 = \Sigma^* = \{a, b\}^*$. In the latter case the set of p.r.f.'s from $\bigcup_{k\geq 0} (\Sigma^*)^k$ to Σ^* can be defined as the smallest set of functions such that it includes:

(i.1) the functions of k arguments $\lambda s_1 \ldots s_k.\, \varepsilon$, where ε is the empty string, for all $k \geq 0$,

(i.2) the two unary successor functions $\lambda s.\, sa$ and $\lambda s.\, sb$, and

(i.3) the projection functions $\lambda s_1 \ldots s_k . s_i$ for all $k \geq 1$ and for all $i = 1, \ldots, k$,

and it is closed under

(ii.1) composition, defined as in Definition 1 (iv),

(ii.2) primitive recursion defined as follows:
if the functions h_a and h_b of $k{+}1$ arguments and a function g of $k{-}1$ arguments are p.r.f.'s for some $k \geq 1$, then also the function f of k arguments such that:

$$\begin{aligned} f(\varepsilon, s_2, \ldots, s_k) &= g(s_2, \ldots, s_k) \\ f(s_1 a,\, s_2, \ldots, s_k) &= h_a(s_1, s_2, \ldots, s_k,\; f(s_1, s_2, \ldots, s_k)) \\ f(s_1 b,\, s_2, \ldots, s_k) &= h_b(s_1, s_2, \ldots, s_k,\; f(s_1, s_2, \ldots, s_k)) \end{aligned}$$

is a p.r.f., and

(ii.3) minimalization in the sense that given any *total* function p.r.f. $\lambda xs.\, t(x, s)$ from $(\Sigma^*)^k \times \Sigma^*$ to Σ^*, then the p.r.f. of arity k of the form

$$\lambda x.(\mu s.[t(x, s) = \varepsilon])$$

where $x \in (\Sigma^*)^k$ and $s \in \Sigma^*$, belongs to the set of p.r.f.'s from $(\Sigma^*)^k$ to Σ^*. In order to apply the minimalization operator μ we need an ordering among the strings in Σ^* and we choose the canonical one, that is, $\varepsilon < a < b < aa < ab < ba < bb < aaa < \ldots$, which corresponds to the assumption that $a < b$.

It is not difficult to show that under suitable encodings of strings into numbers, the theory of the p.r.f.'s where each function is from $(\Sigma^*)^k$ to Σ^*, for some $k \geq 0$, is isomorphic to the theory of the p.r.f.'s where each function is from N to N.

17 Partial Recursive Functions and Turing Computable Functions

We have the following theorem which identifies the set of partial recursive functions and the set of Turing computable functions, when we represent the input/output numbers and the input/output strings as indicated in Definitions 9 and 10 of Section 2. In particular, any natural number n is represented by 01^n0 and for any $k \geq 0$, any k-tuple $\langle n_1, \ldots, n_k \rangle \in N^k$ is represented by $01^{n_1}0 \ldots 01^{n_k}0$.

Theorem 7. (i) For any $k\,(\geq 0)$, any Turing Machine M whose initial configuration is $q_0 0\,1^{n_1}\,0 \ldots 0\,1^{n_k}\,0$, computes a partial recursive function from N^k to N, and any partial recursive function from N^k to N can be computed by a Turing Machine whose initial configuration is $q_0 0\,1^{n_1}\,0 \ldots 0\,1^{n_k}\,0$.

(ii) For any $k\,(\geq 0)$ any Turing Machine M whose initial configuration is $q_0 w$ with $w \in (\Sigma_1^*)^k$, computes a partial recursive function f from $(\Sigma_1^*)^k$ to Σ_2^*, and any partial recursive function from $(\Sigma_1^*)^k$ to Σ_2^* can be computed by a Turing Machine whose initial configuration is $q_0 w$.

We do not give the proof of this theorem here. Theorem 7 tells us that there exists a bijection between partial recursive functions and Turing Machines.

Theorem 7 also holds if instead of Turing Machines and partial recursive functions, we consider, respectively: (i) Turing Machines which always halt with a configuration which represents an output value according to our conventions, and (ii) partial recursive functions which are total. Thus, the notion of a total partial recursive function is equivalent to the notion of an 'always terminating' Turing Machine.

As we will see later, a consequence of this fact is that the undecidability of the Halting Problem implies that there is no partial recursive function which given any partial recursive function f, tells us whether or not f is total.

18 Subrecursive Classes of Partial Recursive Functions

In this section we introduce some classes of partial recursive functions which define three hierarchies within the class PR of the primitive recursive functions (see Section 12 on page 57).

We will first introduce: (i) the Grzegorczyk hierarchy [12], (ii) the loop hierarchy [28], and finally, (iii) the Axt hierarchy [4]. At the end of the section we will relate these three hierarchies.

Let us begin by introducing the Grzegorczyk hierarchy [12]. We first need the following two definitions.

Definition 14. **[Spine Function]** For any $n \geq 0$, the n-th *spine function* f_n: $N \times N \rightarrow N$ is defined as follows:

$f_0(x,y) = x+1$ (*successor*)

$f_1(x,y) = x+y$ (*sum*)

$f_2(x,y) = x \times y$ (*multiplication*)

$f_n(x,y) = \mathit{if}\ y=0\ \mathit{then}\ 1\ \mathit{else}\ f_{n-1}(x, f_n(x, y-1))$ for any $n > 2$.

We have that:

$f_3(x,y) = x^y$ (*exponentiation*)

$f_4(x,y) = \mathit{if}\ y=0\ \mathit{then}\ 1\ \mathit{else}\ x^{f_4(x,\,y-1)}$

Thus, for $y \geq 1$, $f_4(x,y) = x^{\cdot^{\cdot^{\cdot^{x^{x}}}}}$, where x occurs y times on the right hand side, that is, $f_4(x,y) = x \uparrow (x \uparrow \ldots (x \uparrow x) \ldots)$, where the symbol ↑ denotes exponentiation and x occurs y times on the right hand side.

Proposition 1. For all $n \geq 0$, for all $x > 2$, and for all $y > 2$, $f_n(x,y) < f_{n+1}(x,y)$.

Definition 15. **[Limited Primitive Recursion]** We say that a function $f : N^k \rightarrow N$, for $k \geq 1$, is defined by *limited primitive recursion* iff: (i) f is defined by primitive recursion (given two other primitive recursive functions, say g and h), and (ii) there exists a primitive recursive function $h_\ell : N^k \rightarrow N$ such that for all $x_1, \ldots, x_k \in N$ we have that:

$f(x_1, \ldots, x_k) \leq h_\ell(x_1, \ldots, x_k)$.

Definition 16. **[Base Function]** The set of the *base functions* is the smallest class of functions which includes:
(i) the *zero* functions z_k of k arguments, that is, $\lambda x_1 \ldots x_k.0$, for all $k \geq 0$,
(ii) the *successor* function s of one argument, that is, $\lambda x.x+1$, and
(iii) the i-th *projection* functions π_{ki} of k arguments, that is, $\lambda x_1 \ldots x_k.x_i$, for all $k \geq 1$ and all i such that $1 \leq i \leq k$.

Now we will define the n-th Grzegorczyk class of functions, for any $n \geq 0$.

Definition 17. **[Grzegorczyk Class]** For any $n \geq 0$, the n-th Grzegorczyk class of functions, denoted E_n, is the smallest class of functions which includes:
(i) the *base functions*,
(ii) the n-th *spine function* f_n, and it is closed under
(iii) *composition* and
(iv) *limited primitive recursion* using functions in E_n, that is, the functions g, h, and h_ℓ used in Definition 15, belong to E_n.

We have that $\lambda x.x \mathbin{\neg} 1$ (i.e., the predecessor function) and $\lambda x, y.x \mathbin{\neg} y$ (i.e., the proper subtraction function) belong to E_0 (see Section 12 on page 59). We also have that the functions $\lambda x, y.\mathit{quot}(x,y)$ (i.e., the integer quotient when x is divided by y) and $\lambda x, y.\mathit{rem}(x,y)$ (i.e., the integer remainder when x is divided by y) belong to E_0.

We have the following theorem which states that the Grzegorczyk classes of functions form a hierarchy which covers the class PR of the primitive recursive functions.

Theorem 8. [Grzegorczyk (1953)] (i) For all $n \geq 0$, $E_n \subset E_{n+1}$, and (ii) $\bigcup_{n \geq 0} E_n = \mathrm{PR}$.

One can say that the Grzegorczyk hierarchy stratifies the class PR of the primitive recursive functions by *growth complexity* because the spine functions denote different degrees of growth as stated by Proposition 1.

In connection with the results of Proposition 1 and Theorem 8 the reader may also take into consideration Theorem 21 of Section 29.4 which tells us that by using a number of moves which is a primitive recursive function of the size of the input, a Turing Machine can compute only a primitive recursive function of the input.

Let us now introduce a different stratification of the class PR of the primitive recursive functions. It is based on the so called *loop complexity* [28]. We need the following definition.

Definition 18. **[Loop Program]** Let us consider a denumerable set V of variables ranged over by $x, y, \ldots$ A *loop program* p_0 of level 0 is defined as follows:

$$p_0 \ ::= \ x := 0 \mid x := x+1 \mid x := y \mid p_0 \, ; p_0$$

where ';' denotes the usual concatenation of programs.
For any $n > 0$, a loop program p_n of level n is defined as follows:

$$p_n \ ::= \ x := 0 \mid x := x+1 \mid x := y \mid p_n \, ; p_n \mid \textbf{while } x \geq 0 \textbf{ do } p_{n-1} \, ; x := x-1 \textbf{ od}$$

where on the right hand side: (i) p_{n-1} is a loop program of level $n-1$, and (ii) in the program p_{n-1} the variable x cannot occur on the left hand side of any assignment. The class of loop programs of level n, for any $n \geq 0$, is denoted by L_n.

The semantics of the loop programs, which we will not formally define, is the familiar Pascal-like semantics. Note that the value which the variable x has when the execution of a **while** $x \geq 0$ **do** $p\, ; x := x-1$ **od** construct begins, tells us how many times the loop program p will be executed. By using new variable names, we can always write a loop program so that a distinct variable controls the execution of the body of each of its **while-do** constructs.

With every loop program p we can associate a function from N^k to N for some $k \geq 0$, by assuming as inputs the initial values of k distinct variables of p, and as output, the final value of a variable of p.

By abuse of notation, for any $n \geq 0$, we will denote by L_n also the class of functions, called *loop functions*, computed by the loop programs in L_n.

We have the following theorem which states that the loop functions form a hierarchy which covers the class PR of the primitive recursive functions.

Theorem 9. (i) For all $n \geq 0$, $L_n \subset L_{n+1}$, that is, the class of functions computed the loop programs in L_n is properly contained in the class of functions computed by the loop programs in L_{n+1}, (ii) $\bigcup_{n \geq 0} L_n = \mathrm{PR}$, and (iii) for all $n \geq 2$, $L_n = E_{n+1}$.

We have the following decidability and undecidability results.

The equivalence of loop programs within the class L_1 is decidable (that is, given two loop programs in L_1 it is decidable whether or not they compute the same function), while for any $n \geq 2$, the equivalence of loop programs is undecidable.

Let us now introduce one more stratification of the class PR of the primitive recursive functions. It is based on the so called *structural complexity* [4]. We need the following definition.

Definition 19. **[Axt Class]** The Axt class K_0 of functions is the smallest class of functions which includes:
(i) the *base functions*, and it is closed under
(ii) *composition*.

For any $n \geq 0$, the Axt class K_{n+1} of functions is the smallest class of functions which:
(i) includes the class K_n,
(ii) is closed by *composition*, and
(iii) is closed by *primitive recursion* using functions in K_n, that is, the two functions which are required for defining a new function by primitive recursion should be in K_n.

We have the following theorem which states that the Axt classes form a hierarchy which covers the class PR of the primitive recursive functions.

Theorem 10. (i) For all $n \geq 0$, $K_n \subset K_{n+1}$, (ii) $\bigcup_{n\geq 0} K_n = \text{PR}$, and (iii) for all $n \geq 2$, $K_n = E_{n+1}$.

Let us introduce the following class of functions.

Definition 20. [Kalmár Elementary Function] The class E of the *Kalmár elementary functions* is the smallest class of functions which includes:
(i) the *base functions*,
(ii) the *sum* function $\lambda x, y.x + y$, and it is closed under
(iii) *composition*,
(iv) *bounded sum* and *bounded product* (see Definition 6 on page 64).

The usual functions of sum, subtraction, product, division, exponential, and logarithm, whose definitions are suitably modified for ensuring that they are defined in $N \times N$ (or N) and return a value in N (see Section 12 on page 59), belong to the class E of the Kalmár elementary functions. We have the following result.

Theorem 11. The class E of the Kalmár elementary functions coincides with the Grzegorczyk class E_3.

Thus, by Theorems 9 and 10, we also have that: E is equal to: (i) the functions computed by the class L_2 of loop programs, and (ii) the Axt class K_2 (because $E_3 = L_2 = K_2$).

The relationships among the Grzegorczyk classes, the functions computed by the classes of loop programs, and the Axt classes are depicted in Figure 22. We have that:
(i) $K_0 = L_0 \subset E_0$,
(ii) $K_1 \subset L_1 \subset E_1$, and as already stated,
(iii) for any $n \geq 2$, $L_n = K_n = E_{n+1}$.

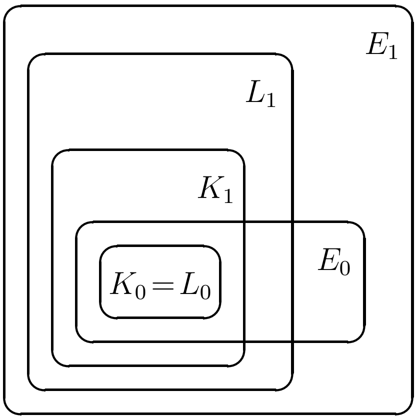


Fig. 22. A pictorial view of the relationships among various classes of functions: (i) the Grzegorczyk classes E_0 and E_1, (ii) the functions computed by the classes L_0 and L_1 of loop programs, and (iii) the Axt classes K_0 and K_1.

Before ending this section let us introduce in the following Definition 21 a new, very powerful technique of defining total partial recursive functions, starting from some other given total partial recursive functions. This technique is parameterized by a natural number n which is assumed to be greater than or equal to 1. In case $n = 1$ this technique coincides with the primitive recursion schema we have introduced in Definition 1 of Section 12.

For reasons of simplicity, in the following definition we have written the single variable y, instead of a tuple of $t\,(\geq 0)$ variables $y_1, \ldots, y_t$.

Definition 21. [Primitive Recursion of Order n] [31, page 119] A function f of $n+1$ arguments from N^{n+1} to N is said to be defined by *primitive recursion of order $n\,(\geq 1)$* starting from the functions $g : N \rightarrow N$, $h : N^{2n+1} \rightarrow N$, and the functions $g_{11}, \ldots, g_{1\,n-1}$, $g_{21}, \ldots, g_{2\,n-2}, \ldots, g_{n-2\,1}, g_{n-2\,2}$, $g_{n-1\,1}$, all from N^{n+2} to N, iff

$f(x_1, \ldots, x_n, y) = g(y) \quad$ *if* $x_1\!=\!0$ *or* $\ldots$ *or* $x_n\!=\!0$

$f(x_1+1, \ldots, x_n+1, y) = h(x_1, \ldots, x_n, y, F_1, \ldots, F_n)$

where, for $j = 1, \ldots, n$,

F_j stands for $f(x_1+1, \ldots, x_{j-1}+1, x_j, y, H_{j1}, \ldots, H_{j\,n-j})$

where, for $p = 1, \ldots, n-j$,

H_{jp} stands for $g_{jp}(x_1, \ldots, x_n, y, f(x_1+1, \ldots, x_{n-1}+1, x_n, y))$.

Primitive recursion of order n is called *n-fold nested recursion* in [31]. For instance, the following function f is defined by primitive recursion of order 1 from the functions g and h (the variable x stands for the variable x_1 of Definition 21):

$f(0, y) = g(y)$

$f(x\!+\!1, y) = h(x, y, f(x, y))$.

This confirms that for $n\!=\!1$ primitive recursion of order 1 coincides with the primitive recursion schema we have introduced in Definition 1 of Section 12.

The following function f is defined by primitive recursion of order 2 from the functions g, h, and k (the variables x, y, and z stand, respectively, for the variables x_1, x_2, and y of Definition 21):

$f(x, y, z) = g(z) \qquad\qquad$ *if* $x\!=\!0$ *or* $y\!=\!0$

$f(x\!+\!1, y\!+\!1, z) = h(x, y, z, F_1, F_2)$,

where F_1 is $f(x,\ z,\ k(x, y, z, f(x\!+\!1, y, z)))$ and F_2 is $f(x\!+\!1, y, z)$.

Note that for any $n\,(\geq 1)$ the schema of primitive recursion of order m, for any $m = 1, \ldots, n\!-\!1$, is a particular case of the schema of primitive recursion of order n.

Definition 22. [Primitive Recursive Function of Order n] Let the class R_n, with $n \geq 1$, of functions be the smallest class of functions which includes:
(i) the *base functions*, and it is closed under
(ii) *composition*, and
(iii) *primitive recursion of order n*.

In Section 13 we have seen that the Ackermann function is *not* primitive recursive. We leave it to the reader to check that it is in R_2 [31, page 119].

We have the following result.

Theorem 12. (i) The class R_1 is the class PR of the primitive recursive functions, (ii) for all $n \geq 0$, $R_n \subset R_{n+1}$, and (iii) $\bigcup_{n\geq 0} R_n$ is *properly contained* in the set of all total PRF's, that is, the set of all partial recursive functions which are total functions.

Point (i) follows from the fact that primitive recursion is equal to primitive recursion of order 1. Point (iii) of the above theorem is a consequence of the fact the set of all total PRF's is *not* recursive enumerable. Indeed, if $\bigcup_{n\geq 0} R_n$ were equal to the set of all total PRF's we would have a way of enumerating all total PRF's (one total p.r.f. for each way of applying the rules of constructing a new total p.r.f. from old p.r.f.'s).

Chapter 3

Decidability

19 Universal Turing Machines and Undecidability of the Halting Problem

In this chapter we need some basic notions about Turing Machines which we have introduced in Section 2. In particular, we need the notions of a Turing Machine which: (i) answers 'yes' (or accepts) a given word w, or (ii) answers 'no' (or rejects) a given word w (see Definition 8 on page 14).

Now let us introduce the following definitions.

Definition 1. [Recursively Enumerable Language] Given an alphabet Σ, we say that a language $L \subseteq \Sigma^*$ is *recursively enumerable*, or *r.e.*, or L is a *recursive enumerable subset* of Σ^*, iff there exists a Turing Machine M such that for all words $w \in \Sigma^*$, M accepts the word w iff $w \in L$.

If a language $L \subseteq \Sigma^*$ is r.e. and M is a Turing Machine that accepts L, we have that for all words $w \in \Sigma^*$, if $w \notin L$ then *either* (i) M rejects w *or* (ii) M 'runs forever' without accepting w, that is, for all configurations γ such that $q_0 w \to_M^* \gamma$, where $q_0 w$ is the initial configuration of M, there exists a configuration γ' such that: (ii.1) $\gamma \to_M \gamma'$ and (ii.2) the states in γ and γ' are *not* final.

Recall that the language accepted by a Turing Machine M is denoted by $L(M)$.

Given any recursively enumerable language L, there are $|N|$ Turing Machines each of which accepts (or recognizes) L. (As usual, by $|N|$ we have denoted the cardinality of the set N of the natural numbers.)

Given the alphabet Σ, we denote by R.E. the class of the recursive enumerable languages subsets of Σ^*.

Definition 2. [Recursive Language] We say that a language $L \subseteq \Sigma^*$ is *recursive*, or L is a *recursive subset* of Σ^*, iff there exists a Turing Machine M such that: (i) for all words $w \in \Sigma^*$, M accepts the word w iff $w \in L$, and (ii) M rejects the word w iff $w \notin L$.

Given the alphabet Σ, we denote by REC the class of the recursive languages subsets of Σ^*. One can show that the class of recursive languages is properly contained in the class of the r.e. languages. Thus, if a language L is not r.e. then L is not recursive.

In the sequel when we say that a language L is r.e., either (i) we mean that L is r.e. and it is not recursive, or (ii) we mean that L is r.e. and we say nothing about L

being recursive or not. The reader should be careful in distinguishing between these two meanings.

Now we introduce the notion of a decidable problem. Together with that notion we also introduce the related notions of a semidecidable problem and an undecidable problem. We first introduce the following three notions.

Definition 3. [Problem, Instance of a Problem, Solution of a Problem] Given an alphabet Σ, (i) a *problem* is a language $L \subseteq \Sigma^*$, (ii) an *instance of a problem* $L \subseteq \Sigma^*$ is a word $w \in \Sigma^*$, and (iii) a *solution of a problem* $L \subseteq \Sigma^*$ is an algorithm, that is, a Turing Machine, which accepts the language L (see Definition 8 on page 14).

Given a problem L, we will is also say that L is the language associated with that problem.

As we will see below (see Definitions 4 and 5), a problem L is said to be decidable or semidecidable depending on the properties of the Turing Machine, if any, which provides a solution of L.

Note that an instance $w \in \Sigma^*$ of a problem $L \subseteq \Sigma^*$ can be viewed as a membership question of the form: «Does the word w belong to the language L?». For this reason in some textbooks a problem, as we have defined it in Definition 3 above, is said to be a *yes-no problem*, and the language L associated with a yes-no problem is also called the *yes-language* of the problem. Indeed, given a problem L, its yes-language which is L itself, consists of all words w such that the answer to the question: «Does w belong to L?» is 'yes'. The words of the yes-language L are called *yes-instances* of the problem.

We introduce the following definitions.

Definition 4. [Decidable and Undecidable Problem] Given an alphabet Σ, a problem $L \subseteq \Sigma^*$ is said to be *decidable* (or *solvable*) iff L is recursive. A problem is said to be *undecidable* (or *unsolvable*) iff it is not decidable.

As a consequence of this definition, every problem L such that the language L is finite, (that is, every problem which has a finite number of instances) is decidable.

Definition 5. [Semidecidable Problem] A problem L is said to be *semidecidable* (or *semisolvable)* iff L is recursive enumerable.

We have that the class of decidable problems is properly contained in the class of the semidecidable problems, because for any fixed alphabet Σ, the recursive subsets of Σ^*, are a proper subset of the recursively enumerable subsets of Σ^*.

Now, in order to fix the reader's ideas, we present two problems: (i) the *Primality Problem*, and (ii) the *Parsing Problem.*

Example 1. **[Primality Problem]** The Primality Problem is the subset of $\{1\}^*$ defined as follows:

Prime $= \{1^n \mid n \text{ is a prime number}\}$.

An instance of the Primality Problem is a word of the form 1^n, for some $n \geq 0$. A Turing Machine M is a solution of the Primality Problem iff for all words of the form 1^n with

$n \geq 1$, we have that M accepts w iff $1^n \in \mathit{Prime}$. Obviously, the yes-language of the Primality Problem is *Prime*. We have that the Primality Problem is decidable.

Note that we may choose other ways of encoding the prime numbers, thereby getting other equivalent ways of presenting the Primality Problem. □

Example 2. **[Parsing Problem]** The Parsing Problem is the subset *Parse* of $\{0,1\}^*$ defined as follows:

$\mathit{Parse} = \{[G]\,000\,[w] \mid w \in L(G)\}$

where $[G]$ is the encoding of a grammar G as a string in $\{0,1\}^*$ and $[w]$ is the encoding of a word w as a string in $\{0,1\}^*$, as we now specify.

Let us consider a grammar $G = \langle V_T, V_N, P, S\rangle$. Let us encode every symbol of the set $V_T \cup V_N \cup \{\rightarrow\}$ as a string of the form 01^n for some value of n, with $n \geq 1$, so that two distinct symbols have two different values of n. Thus, a production of the form: $x_1 \ldots x_m \rightarrow y_1 \ldots y_n$, for some $m \geq 1$ and $n \geq 0$, with the x_i's and the y_i's in $V_T \cup V_N$, will be encoded by a string of the form: $01^{k_1}01^{k_2} \ldots 01^{k_p}0$, where $k_1, k_2, \ldots, k_p$ are positive integers and $p = m+n+1$. The set of productions of the grammar G can be encoded by a string of the form: $0\sigma_1 \ldots \sigma_t 0$, where each σ_i is the encoding of a production of G. Two consecutive 0's denotes the beginning and the end of the encoding of a production. Then $[G]$ can be taken to be the string $01^{k_a}0\sigma_1 \ldots \sigma_t 0$, where 01^{k_a} encodes the axiom of G. We also stipulate that a string in $\{0,1\}^*$ which does *not* comply with the above encoding rules is the encoding of a grammar which generates the empty language.

The encoding $[w]$ of a word $w \in V_T^*$ as a string in $\{0,1\}^*$ is $01^{k_1}01^{k_2} \ldots 01^{k_q}0$, where $k_1, k_2, \ldots, k_q$ are positive integers.

An instance of the Parsing Problem is a word of the form $[G]\,000\,[w]$, where: (i) $[G]$ is the encoding of a grammar G, and (ii) $[w]$ is the encoding of a word $w \in V_T^*$.

A Turing Machine M is a solution of the Parsing Problem if given a word of the form $[G]\,000\,[w]$ for some grammar G and word w, we have that M accepts w iff $w \in L(G)$, that is, M accepts w iff $[G]\,000\,[w] \in \mathit{Parse}$.

Obviously, the yes-language of the Parsing Problem is *Parse.*

We have the following decidability results if we restrict the class of the grammars we consider in the Parsing Problem. In particular,
(i) if the grammars of the Parsing Problem are type 1 grammars then the Parsing Problem is decidable, and
(ii) if the grammars which are considered in the Parsing Problem are type 0 grammars then the Parsing Problem is semidecidable and it is undecidable. □

Care should be taken when considering the encoding of the instances of a yes-no problem. We will not discuss in detail this issue here. We only want to remark that the encoding has to be constructive (i.e., computable by a Turing Machine which always halts), and it should not modify the solution of the problem itself (and this could happen when the problem is of syntactic nature as, for instance, when it depends on the way in which a problem is formalized in a particular language).

Definition 6. [Property Associated with a Problem] With every problem $L \subseteq \Sigma^*$ for some alphabet Σ, we associate a *property* P_L such that $P_L(x)$ holds iff $x \in L$.

For instance, in the case of the Parsing Problem, $P_{Parsing}(x)$ holds iff x is a word in $\{0,1\}^*$ of the form $[G]000[w]$, for some grammar G and some word w such that $w \in L(G)$.

Instead of saying that a problem L is decidable (or undecidable, or semidecidable, respectively), we will also say that the associated property P_L is decidable (or undecidable, or semidecidable, respectively).

Remark 1. [**Specifications of a Problem**] As it is often done in the literature, we will also specify a problem $\{x \mid P_L(x)\}$ by using the sentence:

«Given x, determine whether or not $P_L(x)$ holds»

or by asking the question:

«$P_L(x)$?»

Thus, for instance, (i) instead of saying 'the problem $\{x \mid P_L(x)\}$', we will also say 'the problem of determining, given x, whether or not $P_L(x)$ holds', and (ii) instead of saying 'the problem of determining, given a grammar G, whether or not $L(G) = \Sigma^*$ holds', we will also ask the question '$L(G) = \Sigma^*$?' (see the entries of Figure 25 on page 105 and Figure 28 on page 112). □

In what follows we need the following facts and theorem.

(i) The complement $\Sigma^* - L$ of a recursive set L is recursive.
In order to show this, it is enough to use the Turing Machine M for recognizing L. Given any $w \in \Sigma^*$, if M halts and answers 'yes' (or 'no') for w then the Turing Machine $M1$ which recognizes $\Sigma^* - L$, halts and answers 'no' (or 'yes', respectively) for w.

(ii) The union of two recursive languages is recursive. The union of two r.e. languages is r.e. (We leave it to the reader to construct the Turing Machines which justify these facts.)

Theorem 1. [**Post Theorem**] If a language L and its complement $\Sigma^* - L$ are r.e. languages, then L is recursive.

Proof. It is enough to have the Turing Machine, say $M1$, for recognizing L, and the one, say $M2$, for recognizing Σ^*–L, running in parallel so that they perform their computations in an interleaved way. This means that, given an input word $w \in \Sigma^*$, initially $M1$ makes one computation step, then $M2$ makes one computation step, then $M1$ makes one more computation step, and so on, in an alternate way. One of the two machines will eventually halt (because given a word $w \in \Sigma^*$ either $w \in L$ or $w \notin L$), and it will give us the required answer. For instance, if it is $M1$ (or $M2$) which halts first and answers 'yes' for w, then the answer is 'yes' (or 'no', respectively) for w. □

Thus, given any set $L \subseteq \Sigma^*$ there are four mutually exclusive possibilities only (see also the following Figure 23):

(i)	L is recursive	and $\Sigma^* - L$ is recursive
(ii)	L is not r.e.	and $\Sigma^* - L$ is not r.e.
(iii.1)	L is r.e. and not recursive	and $\Sigma^* - L$ is not r.e.
(iii.2)	L is not r.e.	and $\Sigma^* - L$ is r.e. and not recursive

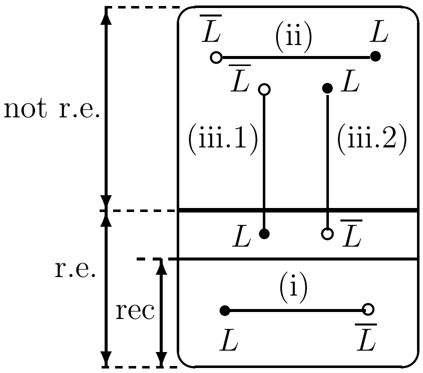


Fig. 23. Post Theorem: view of the set of subsets of Σ^*. $\bullet$ denotes a language L subset of Σ^*, and $\circ$ denotes the complement language $\overline{L} =_{def} \Sigma^* - L$. (i) L and $\overline{L}$ are both recursive. (ii) L and $\overline{L}$ are both not r.e. (iii.1) L is r.e. and $\overline{L}$ is not r.e. (iii.2) L is not r.e. and $\overline{L}$ is r.e.

As a consequence, in order to show that a problem is unsolvable and its associated language L is not recursive, it is enough to show that $\Sigma^* - L$ is not r.e.

Another technique for showing that a language is not recursive (and, thus, the associated problem is undecidable) is the *reduction technique.*

We say that a problem A whose associated yes-language is L_A, subset of Σ^*, is *reduced to* a problem B whose associated yes-language is L_B, also subset of Σ^*, iff there exists a total function, say r, from L_A to L_B such that for every word w in Σ^*, w is in L_A iff $r(w)$ is in L_B. (Note that r need not be a surjection.) Thus, if the problem B is decidable then the problem A is decidable, and if the problem A is undecidable then the problem B is undecidable.

Let us now consider a yes-no problem, called the *Halting Problem.* This problem is formulated as follows. Given a Turing Machine M and a word w, it should be determined whether or not M halts on the input w. The various instances of this problem are given by the instances of the machine M and the word w.

We recall that given a language $R \subseteq \{0,1\}^*$ recognized by some Turing Machine M, there exists a Turing Machine $M1$ with tape alphabet $\{0,1,B\}$ such that R is recognized by $M1$.

Thus, if we prove that the Halting Problem is undecidable for Turing Machines with tape alphabet $\{0,1,B\}$ and words in $\{0,1\}^*$, then the Halting Problem is undecidable for any class of Turing Machines and words which includes this one.

Definition 7. [Binary Turing Machines] A Turing Machine is said to be *binary* iff its tape alphabet is $\{0,1,B\}$ and its input words belong to $\{0,1\}^*$.

We will show that the Halting Problem for binary Turing Machines with one tape is undecidable. In order to do so, we first show that a Turing Machine M with tape alphabet $\{0,1,B\}$ can be encoded by a word in $\{0,1\}^*$. Without loss of generality, we may assume that M is deterministic.

Let us assume that:
- the set of states of M is $\{q_i \mid 1 \leq i \leq n\}$, for some value of $n \geq 2$,
- the tape symbols 0, 1, and B are denoted by X_1, X_2, and X_3, respectively, and
- L (that is, the move to the left) and R (that is, the move to the right) are denoted by 1 and 2, respectively.

The initial and final states are assumed to be q_1 and q_2, respectively. Without loss of generality, we may also assume that there exists one final state only.

Then, each quintuple 'q_i, $X_h \rightarrow q_j$, X_k, m' of M corresponds to a string of five positive numbers $\langle i, h, j, k, m \rangle$. It should be the case that $1 \leq i, j \leq n$, $1 \leq h, k \leq 3$, and $1 \leq m \leq 2$. Thus, the quintuple 'q_i, $X_h \rightarrow q_j$, X_k, m' can be encoded by the sequence: $1^i 0 1^h 0 1^j 0 1^k 0 1^m$.

The various quintuples can be listed one after the other, in any order, so to get a sequence of the form:

000 code of the first quintuple 00 code of the second quintuple 00 ... 000. (†)

Every sequence of the form (†) encodes one Turing Machine only, while a Turing Machine can be encoded by several sequences (recall that, for instance, when describing a Turing Machine the order of the quintuples is not significant).

We assume that the sequences of 0's and 1's which are not of the form (†) are encodings of a Turing Machine which recognizes the empty language, that is, a Turing Machine which does not accept any word in $\{0, 1\}^*$.

Let us now consider the following language subset of $\{0, 1\}^*$:

$L_U = \{[M]w \mid \text{the Turing Machine } M \text{ accepts the word } w\}$

where $[M]$ is the encoding of M as a word in $\{0, 1\}^*$, w is a word in $\{0, 1\}^*$, and $[M]w$ is the concatenation of $[M]$ and w. (The subscript U in L_U stands for 'universal'.) The leftmost symbol of the word w is immediately to the right of the three 0's in the rightmost position of $[M]$.

Theorem 2. [Turing Theorem. Part 1] L_U is recursive enumerable.

Proof. We show that L_U is recursively enumerable by showing that there exists a binary Turing Machine M_U which recognizes L_U.

Indeed, we show the existence of a Turing Machine, say T, with three tapes, called Tape1, Tape2, and Tape3, respectively, which recognizes L_U. Then from T we construct the equivalent binary Turing Machine M_U, called *Universal Turing Machine*, having one one-way infinite tape and 0, 1, and B as tape symbols.

The machine T works as follows.

Initially T has in Tape1 a sequence s in $\{0, 1\}^*$. Then T checks whether or not s is equal to $s_1 s_2$ for some s_1 of the form (†).

If this is not the case, then T halts without entering a final state, and this indicates that M does not accept w (recall that we have assumed that a sequence not of the form (†) encodes a Turing Machine which recognizes the empty language).

Otherwise, if the sequence s can be viewed as the concatenation of a sequence of the form (†) which encodes a Turing Machine, say M, followed by a word w in $\{0, 1\}^*$, then T writes w in its Tape2 and '1' in its Tape3 (this '1' denotes that M starts its computation in its initial state q_1). The machine T continues working by simulating the behaviour of the machine M whose quintuples are on Tape1, on the input word w which is on Tape2, storing the information about the states of M on Tape3. In order to recall which symbol of the word w is currently scanned by M, the machine T uses a second track of Tape2. The only final state of M is q_2, which is encoded by 11.

(α) Knowing the current state q_i and the scanned symbol X_h, the machine T checks whether or not 11 (denoting the accepting state q_2) is on the Tape3. If this is the case, then T stops and accepts the word w. If this is *not* the case, then T looks for a quintuple on the first tape of the form: $001^i01^h01^j01^k01^m00$, and

(α1) if there is one quintuple of that form then T makes the corresponding move by: (i) placing 1^j on Tape3, (ii) writing 0 or 1 on the scanned cell of Tape2, if the value of k is 1 or 2, respectively, and (iii) changing the head marker on the second track of Tape2 to the left or to the right of the current one, if the value of m is 1 or 2, respectively. Then, T continues working as specified at Point (α) looking at the current state on Tape3.

(α2) If it does *not* exist a quintuple of the form $001^i01^h01^j01^k01^m00$ then T stops and it does not accept the input word because 11 is not on Tape3.

It should be clear from the above construction that the machine T simulates the machine M on the input w in the sense that: (i) T halts and accepts $[M]w$ iff M halts and accepts w, (ii) T halts and does not accept $[M]w$ iff M halts and does not accept w, and (iii) T does not halt iff M does not halt. □

Remark 2. Universal Turing Machines are *not* very complex machines. We know that there is universal Turing Machines with 1 tape, 5 states, and 7 tape symbols [16, page 173].

Theorem 3. [Turing Theorem. Part 2] L_U is not recursive.

Proof. We show that L_U is not recursive by showing first that the language L_D, we will define below, is not recursive. (The subscript D in L_D stands for 'diagonal'.) We will show that L_D is not recursive by showing that $\Sigma^* - L_D$ is not r.e. using a diagonal argument.

Then, by reducing L_D to L_U we will derive that L_U is not recursive.

The language L_D is constructed as follows. Let us consider a matrix W whose rows and columns are labeled by the words of $\{0,1\}^*$ in the canonical order, i.e., ε, 0, 1, 00, 01, 10, 11, 000, ... (in this order we have assumed that $0 < 1$). In row i and column j we write '1' if the Turing Machine whose code is the i-th word in the canonical order, accepts the j-th word in the canonical order. (Recall that we have assumed that a sequence not of the form (†) encodes a Turing Machine which recognizes the empty language.) Otherwise we write '0'.

Let L_D be $\{w \mid w \text{ is } i\text{-th word of the canonical order and } W[i,i]=1\} \subseteq \{0,1\}^*$. That is, w belongs to L_D iff the Turing Machine with code w accepts the word w.

We now show that $\Sigma^* - L_D$ is not r.e. Indeed, we have that:
(i) w belongs to $\Sigma^* - L_D$ iff the Turing Machine with code w, does not accept w, and
(ii) if $\Sigma^* - L_D$ is r.e. then there exists a Turing Machine, say M, with code $[M]$, such that M recognizes $\Sigma^* - L_D$.

Let us now ask ourselves whether or not the Turing Machine with code $[M]$, accepts the word $[M]$. We have that the Turing Machine with code $[M]$, accepts the word $[M]$ iff (by definition of $\Sigma^* - L_D$) the Turing Machine whose code is $[M]$ does not accept $[M]$.

This contradiction shows that $\Sigma^* - L_D$ is not r.e. Thus, L_D is not recursive.

Now, in order to get the thesis it is enough to reduce L_D to L_U. We do this as follows. Given a word w in Σ^*, we compute the word ww (the first w plays the role of the code of a Turing Machine, and the second w is the word to be given as input to that Turing Machine). By construction we have that $w \in L_D$ iff $ww \in L_U$.

Since L_D is not recursive, we get that L_U is not recursive. □

We have that the problem corresponding to the language L_U, called the Halting Problem, is not decidable, because L_U is not recursive.

Thus, to say that the Halting Problem is undecidable means that it does not exists any Turing Machine that: (i) halts for every input, and (ii) given the encoding $[M]$ of a Turing Machine M and an input word w for M, answers 'yes' for $[M]w$ iff M halts in a final state when starting initially with w on its tape.

By reduction of the Halting Problem to each of the following problems, one can show that also these problems are undecidable:

(i) *Blank Tape Halting Problem*: given a Turing Machine M, determine whether or not M halts in a final state when its initial tape has blank symbols only,

(ii) *Uniform Halting Problem* (or *Totality Problem*): given a Turing Machine M, determine whether or not M halts in a final state for every input word in Σ^*, and

(iii) *Printing Problem*: given a Turing Machine M and a tape symbol b, determine whether or not M prints b during any of its computations.

Note that it is undecidable whether or not a given binary Turing Machine M ever writes three consecutive 1's on its tape (the various instances of the problem being given by the choice of M), while it is decidable whether or not, given a binary Turing Machine M starting on a blank tape, there exists a cell scanned by M five or more times (the various instances of the problem being given by the choice of M) [16, page 193].

20 Rice Theorem

Let us consider the alphabet $\Sigma = \{0, 1\}$ and the set RE whose elements are the recursive enumerable languages over Σ, i.e., the recursive enumerable subsets of $\{0, 1\}^*$.

Let us consider a subset S of RE. We say that S is a *property* over RE. The property S is said to be *trivial* iff S is empty or S is the whole set RE.

Any subset S of RE corresponds to a yes-no problem and an instance of that problem is given by an r.e. language for which we want to know whether or not it is S. We need an encoding of an r.e. language as a word in $\{0, 1\}^*$. This can be done by using the encoding of the Turing Machine which recognizes that r.e. language. Obviously, an r.e. language is given as input to a Turing Machine via the encoding of the Turing Machine which recognizes it.

The property $\emptyset$ (that is, the empty subset of RE) corresponds to the yes-no problem 'Is a given r.e. language not an r.e. language?' (Obviously, the answer to this question is 'no'), and the property RE corresponds to the yes-no problem 'Is a given r.e. language an r.e. language?' (Obviously, the answer to this question is 'yes').

Now we ask ourselves whether or not a given set S, subset of RE, is itself r.e., i.e., whether or not there exists a Turing Machine M_S corresponding to S, such that given any r.e. set $X \subseteq \Sigma^*$, M_S accepts X iff $X \in S$. We have the following result whose proof uses the notion of the language $L_U \subseteq \Sigma^*$ which we have introduced in Section 19.

Theorem 4. [Rice Theorem for recursive properties over RE] A set S of languages, subset of RE, is recursive iff it is trivial (that is, the yes-no problem of determining whether or not an r.e. language belongs to S is decidable iff $S = \emptyset$ or $S = \text{RE}$).

Proof. Let us consider a non-trivial subset S of languages in RE. We will prove that S is not recursive.

We may consider, without loss of generality, that the empty language (which is obviously an r.e. language) is not in S. Indeed, if the empty language belongs to S then we will first prove that RE$-S$ is not recursive (and we have that the empty language does not belong to RE$-S$). Thus, if RE$-S$ is not recursive we have that S is not recursive either. (Recall that if a set is not recursive, its complement is not recursive.)

Since S is not empty (because S is not trivial) we have that an r.e. language, say L, belongs to S. Let us consider the Turing Machine M_L which recognizes the language L.

The proof of Rice Theorem is by reduction of the Halting Problem (that is, the membership problem for the language $L_U = \{[M]w \mid$ the Turing Machine M accepts the word $w\}$) to the problem of determining whether or not any given r.e. language belongs to S.

This reduction is done by showing that for any given Turing Machine M and any given word w there exists an r.e. language, say A, such that M accepts w iff A belongs to S, that is, for any given Turing Machine M and any given word w there exists a Turing Machine, say N_A, such that M accepts w iff the language accepted by N_A belongs to S.

Indeed, let us consider the following Turing Machine N_A (see also Figure 24 below).

For any given input word v, N_A simulates the Turing Machine M on the input word w. If M accepts w then N_A starts working on the input v as the machine M_L. If M does not accept w then N_A does not accept v. Thus, if M accepts w then N_A accepts L (which is a language in S), and if M does not accept w then N_A accepts no word, that is, N_A accepts the empty language (which is a language not in S). Therefore, M accepts w iff the language accepted by N_A belongs to S.

From the above reduction of the Halting problem to the problem of determining whether or not any given r.e. language A belongs to S, and from the fact that the Halting Problem is undecidable (because the language L_U is not recursive) we get that the set S is not recursive. □

As simple corollaries of Rice Theorem, we have that:

(i) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that $L(M)$ is empty$\}$ is not recursive,

(ii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that $L(M)$ is finite$\}$ is not recursive,

(iii) the set $\{[M] \mid [M]$ is an encoding of a Turing Machine M such that $L(M)$ is regular$\}$ is not recursive, and

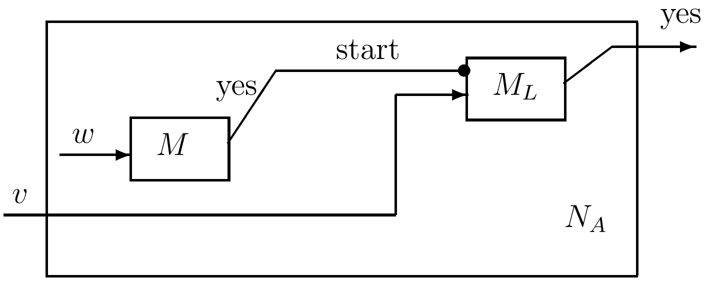


Fig. 24. Construction of the Turing Machine N_A. If M accepts w then N_A accepts the language L (which is in S). If M does not accept w then N_A accepts the empty language (which is not in S).

(iv) the set $\{[M] \,|\, [M]$ is an encoding of a Turing Machine M such that $L(M)$ is context-free$\}$ is not recursive.

We state without proof the following result which is the analogous of the Rice Theorem with respect to the r.e. properties, instead of the recursive properties.

Theorem 5. [Rice Theorem for r.e. properties over RE] A subset S of r.e. languages in RE is r.e. iff S satisfies the following three conditions:
(1) (*r.e. upward closure*) if $L1 \subseteq \{0,1\}^*$ is in S and $L1 \subseteq L2$ for some $L2$ which is r.e., then $L2$ is in S,
(2) (*existential finite downward closure*) if L is an infinite r.e. language in S, then there exists some finite (and thus, r.e.) language L_f subset of L such that L_f is in S, and
(3) the set of the finite languages which are elements of S, is r.e. □

We have the following simple corollaries of Rice Theorem for r.e. properties over RE:

(i) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ is empty$\}$ is *not* r.e. (because condition (1) does not hold),

(ii) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ is $\Sigma^*\}$, is *not* r.e. (because condition (2) does not hold),

(iii) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ is recursive$\}$ is *not* r.e. (because condition (1) does not hold),

(iv) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ is not recursive$\}$ is *not* r.e. (because condition (1) does not hold),

(v) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ is regular$\}$ is *not* r.e. (because condition (1) does not hold),

(vi) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ consists of one word only$\}$ is *not* r.e. (because condition (1) does not hold), and

(vii) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M) - L_U \neq \emptyset\}$ is *not* r.e. (because condition (3) does not hold. Indeed, take a finite set, say D, with empty intersection with L_U. The set $\Sigma^* - L_U$ is a superset of D and yet it is not r.e., because L_U is r.e. and not recursive. Recall also that $L(M) - L_U \neq \emptyset$ is the same as $L(M) \cap (\Sigma^* - L_U) \neq \emptyset$.).

We also have the following corollaries:

(viii) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M) \neq \emptyset\}$ is r.e. (and we also have that it is not recursive),

(ix) for any fixed $k\,(\geq 0)$, the set $S_k = \{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ has at least k elements$\}$ is r.e.,

(x) for any fixed word w, the set $S_w = \{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M)$ has at least the word $w\}$ is r.e., and

(xi) the set $\{[M] \,|\, [M]$ is the encoding of a Turing Machine M such that $L(M) \cap L_U \neq \emptyset\}$ is r.e.

21 Post Correspondence Problem

An instance of the *Post Correspondence Problem*, PCP for short, over the alphabet Σ, is given by two sequences of k words each, say $u = \langle u_1, \ldots, u_k \rangle$ and $v = \langle v_1, \ldots, v_k \rangle$, where the u_i's and the v_i's are elements of Σ^*. A given instance of the PCP has a solution iff there exists a sequence of indexes taken from the set $\{1, \ldots, k\}$, say $\langle i_1, \ldots, i_n \rangle$, for $n \geq 1$, such that the following equality between words of Σ^*, holds:

$$u_{i_1} \ldots u_{i_n} = v_{i_1} \ldots v_{i_n}$$

An instance of the Post Correspondence Problem can be represented in a tabular way as follows:

	u		v
1	u_1	$\rightarrow$	v_1
...	...		...
k	u_k	$\rightarrow$	v_k

and, in this representation, a solution of the instance is given by a sequence of row indexes.

One can show that the Post Correspondence Problem is unsolvable if $|\Sigma| \geq 2$ and $k \geq 2$ by showing that the Halting Problem can be reduced to it.

The Post Correspondence Problem is semisolvable. Indeed, one can find the sequence of indexes which solves the problem, if there exists one, by checking the equality of the words corresponding to any sequence of indexes taken one at a time in the canonical order over the set $\{i \,|\, 1 \leq i \leq k\}$, that is, 1, 2, ..., k, 11, 12, ..., 1k, 21, 22,..., 2k, ...,kk, 111, ..., kkk,...

There exists a variant of the Post Correspondence Problem, called the *Modified Post Correspondence Problem*, where it is required that in the solution sequence, if any, the index i_1 be 1. Also the Modified Post Correspondence Problem is unsolvable and semisolvable.

If the alphabet Σ has one symbol only, then the Post Correspondence Problem is solvable. Indeed, let us denote by $|u|$ the number of symbols of Σ in a word u.

If there exists i, with $1 \leq i \leq k$, such that $u_i = v_i$ then the sequence $\langle i \rangle$ is the solution. If for all i, $1 \leq i \leq k$, $u_i \neq v_i$ then there are two cases: (1) either for all i, $|u_i| > |v_i|$, or for all i, $|u_i| < |v_i|$, and (2) there exist i and j, with $1 \leq i, j \leq k$, such that $|u_i| > |v_i|$ and $|u_j| < |v_j|$.

In Case (1) there is no solution to the Post Correspondence Problem, and in Case (2) there is a solution which is given by the sequence of $p\,(\geq 1)$ i's followed by $q\,(\geq 1)$ j's such that:

$p(|u_i| - |v_i|) + q(|u_j| - |v_j|) = 0.$

We can always take p to be $-(|u_j| - |v_j|)$ and q to be $|u_i| - |v_i|$. Note that both p and q are positive integers.

If $k = 1$ then the Post Correspondence Problem is decidable, and it has a solution iff $u_1 = v_1$.

Example 3. Let us consider the instance of the Post Correspondence Problem over $\{0, 1\}$, given by the following table:

	u		v
1	01	→	11011
2	001	→	00
3	111	→	11

This instance of the Post Correspondence Problem has the solution $\langle 2, 3, 1, 3\rangle$, because: $u_2u_3u_1u_3 = v_2v_3v_1v_3$. Indeed, we have that $u_2u_3u_1u_3 = v_2v_3v_1v_3 = 00111101111$. □

22 Decidability and Undecidability in Formal Languages

In this section we present some decidability and undecidability results about context-free languages. In particular, using the undecidability of the Post Correspondence Problem, we will show (see Theorem 6 on page 99) that it is undecidable whether or not a given context-free grammar G is ambiguous, that is, it is undecidable whether or not there exists a word w such that using the grammar G, the word w has two distinct leftmost derivations.

For the reader's convenience let us first recall a few basic definitions about grammars and formal languages. More information on these subject can be found, for instance, in [16].

Definition 8. [Formal Grammars] A *grammar* (also called a *type* 0 *grammar*) is a 4-tuple $\langle V_T, V_N, P, S\rangle$, where:
- V_T is a countable set of symbols, called *terminal symbols*,
- V_N is a countable set of symbols, called *nonterminal symbols* or *variables*, with $V_T \cap V_N \neq \emptyset$,
- P is a set of pairs of strings, called *productions,* each pair being denoted $\alpha \rightarrow \beta$, where $\alpha \in V^+$ and $\beta \in V^*$, with $V = V_T \cup V_N$, and
- S is an element of V_N, called *axiom* or *start symbol.*

For any string $\sigma \in V^*$, we stipulate that the length of a string σ, denoted *length*(σ) or $|\sigma|$, is n iff $\sigma \in V^n$. A grammar is said to be *context-sensitive* (or *of type* 1) iff for every production $\alpha \rightarrow \beta$ we have that: *either* (i) $\alpha = \gamma A \delta$ and $\beta = \gamma w \delta$, with $\gamma, \delta \in V^*$, $A \in V_N$, and $w \in V^+$, *or* (ii) the production $\alpha \rightarrow \beta$ is $S \rightarrow \varepsilon$, and the axiom S does not occur in the right hand side of any production. A grammar is said to be *context-free* (or *of type* 2) iff for every production $\alpha \rightarrow \beta$, we have that $\alpha \in V_N$. A grammar is said to be *regular* (or *of type* 3) iff for every production $\alpha \to \beta$, we have that $\alpha \in V_N$ and $\beta \in V_T V_N \cup V_T \cup \{\varepsilon\}$, that is, β is either a terminal symbol followed by a nonterminal symbol, or a terminal symbol, or the empty string ε.

The *language* $L(G)$ *generated* by the grammar $G = \langle V_T, V_N, P, S\rangle$ is the set of strings in V_T^* which can be derived from S by successive rewriting according to the productions in P (for a formal definition the reader may refer to [16]). For instance, the language generated by the grammar $\langle \{a,b\}, \{S\}, \{S \to aSb,\ S \to ab\}, S\rangle$ is $\{a^n b^n \mid n \geq 1\}$.

A grammar $G1$ is said to be *equivalent* to a grammar $G2$ iff $L(G1) = L(G2)$.

A symbol X in the set V_N of the nonterminal symbols, is said to be *useful* iff there is a derivation $S \to^* \alpha X \beta \to^* w$ for some $\alpha, \beta \in (V_T \cup V_N)^*$, and some $w \in V_T^*$. X is *useless* iff it is not useful.

A language $L \subseteq V_T^*$ is said to be *context-free* (or *context-sensitive*) *language* iff it is generated by a context-free (or context-sensitive, respectively) grammar. A *deterministic context-free language* is a context-free language which is recognized by a deterministic pushdown automaton. A context-free language L is said to be *inherently ambiguous* iff every context-free grammar generating L is *ambiguous*. A grammar G is said to be *ambiguous* iff there is a word in the language generated by G which has two distinct leftmost derivations using the grammar G. As already mentioned, the reader who is not familiar with these notions will find it useful to look at [16]. In [16] the reader will also find the definitions of the class of $LR(k)$ languages (and grammars), for any $k \geq 0$, which we will consider in the sequel. These notions are related to the deterministic context-free languages (and grammars).

Theorem 6. The ambiguity problem for context-free languages is undecidable.

Proof. It is enough to reduce the Post Correspondence Problem to the ambiguity problem of context-free grammars. Consider a finite alphabet Σ and two sequences of k (≥ 1) words, each word being an element of Σ^*:

$U = \langle u_1, \ldots, u_k\rangle$, and

$V = \langle v_1, \ldots, v_k\rangle$.

Let us also consider the set A of k new symbols $\{a_1, \ldots, a_k\}$ such that $\Sigma \cap A = \emptyset$, the following two languages which are subsets of $(\Sigma \cup A)^*$:

$U_L = \{u_{i_1} u_{i_2} \ldots u_{i_r} a_{i_r} \ldots a_{i_2} a_{i_1} \mid r \geq 1 \text{ and } 1 \leq i_1, i_2, \ldots, i_r \leq k\}$, and

$V_L = \{v_{i_1} v_{i_2} \ldots v_{i_r} a_{i_r} \ldots a_{i_2} a_{i_1} \mid r \geq 1 \text{ and } 1 \leq i_1, i_2, \ldots, i_r \leq k\}$.

A grammar G for generating the language $U_L \cup V_L$ is as follows:
$\langle \Sigma \cup A,\ \{S, S_U, S_V\},\ P,\ S\rangle$, where P is the following set of productions:

$S \to S_U,\ \ S_U \to u_i S_U a_i \mid u_i a_i$ for any $i = 1, \ldots, k$, and

$S \to S_V,\ \ S_V \to v_i S_V a_i \mid v_i a_i$ for any $i = 1, \ldots, k$.

To prove the theorem we now need to show that the instance of the Post Correspondence Problem for the sequences U and V has a solution *iff* the grammar G is ambiguous.
(*only-if part*) If $u_{i_1} \ldots u_{i_n} = v_{i_1} \ldots v_{i_n}$ for some $n \geq 1$, then we have that the word w which is

$u_{i_1} u_{i_2} \ldots u_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$

is equal to the word

$v_{i_1} v_{i_2} \ldots v_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$

and w has two leftmost derivations: one derivation which first uses the production $S \rightarrow S_U$, and a second one, which first uses the production $S \rightarrow S_V$. Thus, G is ambiguous.

(*if part*) Assume that G is ambiguous. Then there are two leftmost derivations for a word generated by G. Since every word generated by S_U has one leftmost derivation only, and every word generated by S_V has one leftmost derivation only (and this is due to the fact that the a_i's symbols force the uniqueness of the productions used when deriving a word from S_U or S_V), it must be the case that a word generated from S_U is the same as a word generated from S_V. This means that we have:

$u_{i_1} u_{i_2} \ldots u_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1} = v_{i_1} v_{i_2} \ldots v_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$

for some sequence $\langle i_1, i_2, \ldots, i_n \rangle$ of indexes with $n \geq 1$, each index being from the set $\{1, \ldots, k\}$. Thus,

$u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}$

and this means that the corresponding Post Correspondence Problem has the solution $\langle i_1, i_2, \ldots, i_n \rangle$. □

Some more decidability and undecidability results about context-free language can be proved using properties of the so-called *valid* and *invalid computations* of Turing Machines. They are introduced in the following definition where by the word w^R we mean the word 'w reversed', that is, for some alphabet A, (i) $\varepsilon^R = \varepsilon$, and (ii) for any $x \in A$ and $w \in A^*$, $(w\,x)^R = x\,w^R$.

Definition 9. [Valid and Invalid Computations of Turing Machines] Given a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where as usual $Q \cap \Gamma = \emptyset$, a string of the form

$w_1 \# w_2^R \# w_3 \# w_4^R \# w_5 \# \ \ldots \ \# w_n$ or

$w_1 \# w_2^R \# w_3 \# w_4^R \# w_5 \# \ \ldots \ \# w_{n-1} \# w_n^R$,

where # is a symbol which is not in $Q \cup \Gamma$, is said to be a *valid computation* of M iff:

(i) each w_i is a configuration of M, that is, a string in $\Gamma^* Q \Gamma^*$, which does not end by the blank symbol B,

(ii) w_1 is the initial configuration, that is, $w_1 = q_0 u$ where u belongs to Σ^*,

(iii) w_n is a final configuration, that is, a string in $\Gamma^* F \Gamma^*$, and

(iv) for each i, with $1 \leq i < n$, $w_i \rightarrow w_{i+1}$.

An *invalid computation* of a Turing Machine is a string which is *not* a Turing Machine valid computation.

We have the following properties.

Property (P1): given a Turing Machine M the set of its valid computations is the intersection of two context-free languages (and the two corresponding context-free grammars can be effectively constructed from M), and

Property (P2): given a Turing Machine M the set of its invalid computations is a context-free language (and the corresponding context-free grammar can be effectively constructed from M).

We have the following theorem.

Theorem 7. (i) It is undecidable whether or not given two context-free grammars $G1$ and $G2$, we have that $L(G1) \cap L(G2) = \emptyset$.
(ii) It is undecidable whether or not given two context-free grammars, they generate the same context-free language.
(iii) It is undecidable whether or not given a context-free grammar, it generates Σ^+ (or Σ^* if one assumes, as we do, that context-free grammars may have the production $S \to \varepsilon$).
(iv) It is undecidable whether or not given a context-free grammar, it generates a deterministic context-free language, that is, it is equivalent to a deterministic context-free grammar.

Proof. See [16]. The proof of (i) is based on the fact that the problem of determining whether or not a given Turing Machine recognizes the empty language, can be reduced to this problem. (Recall the above Property (P1) of the valid computations of a Turing Machine.) □

It is undecidable whether or not a context-sensitive grammar generates a context-free language [3, page 208].

A context-free grammar is said to be *linear* iff the right hand side of each production has at most one nonterminal symbol.

It is undecidable whether or not the language $L(G)$ generated by a context-free grammar G, can be generated by a linear context-free grammar.

A language L is said to be *prefix-free* (or to enjoy the *prefix property*) iff no word in L is a proper prefix of another word in L, that is, for every word $x \in L$ there is no word in L of the form xy with $y \neq \varepsilon$.

It is undecidable whether or not a context-free grammar generates a prefix-free language. Indeed, this problem can be reduced to the problem of checking whether or not two context-free languages have empty intersection [14, page 262]. If we know that the given language is a deterministic context-free language, then this problem is decidable [14, page 355].

It is undecidable whether or not given a context-free grammar G there exist $k\,(\geq 0)$ and an $LR(k)$ grammar $G1$ such that G and $G1$ generate the same language (see [23] and [3, pages 397–399, Exercise 5.2.12]).

This undecidability result can be derived as follows. We have that: (i) the class of the $LR(0)$ languages is contained in the class of the $LR(1)$ languages, and (ii) for any $k > 0$ the class of the $LR(k)$ languages is equal to the class of the $LR(1)$ languages (which is the class of deterministic context free languages). Thus, undecidability holds because, otherwise, it would have been decidable the problem of knowing whether or not a context free grammar is equivalent to a deterministic context-free grammar.

It is undecidable whether or not given a context-free grammar G and a number $k \geq 0$, G equivalent to an $LR(k)$ grammar [23, pages 397–399, Exercise 5.2.12]. For $k > 0$ we can reason as follows. Since for any $k > 0$ the class of the $LR(k)$ languages is equal to the class of the $LR(1)$ languages (which is the class of deterministic context-free languages), this problem is undecidable for $k > 0$ because otherwise, it would have been decidable the problem of testing whether or not a context-free grammar generates a deterministic context-free language.

It is undecidable whether or not given a context-free grammar G there exists $k\,(\geq 0)$ such that G is a $LR(k)$ [38, page 399].

Now we present a theorem which is analogous to Rice Theorem (see Theorem 4 on page 95), but it refers to the class of context-free languages, rather than r.e. languages. We need first the following two definitions.

Definition 10. We say that a class C of languages is *effectively closed under concatenation with regular sets and union* iff for any given language $L1$ and $L2$ and any regular language R, we have that the languages (i) $R \cdot L1$, (ii) $L1 \cdot R$, and (iii) $L1 \cup L2$ are in C, and it is possible to produce the encodings (as a string in $\{0,1\}^*$) of the grammars which generate those three languages from the encodings of any three grammars which generate $L1$, $L2$, and R.

Definition 11. **[Quotient Languages]** Given an alphabet Σ, a language $L \subseteq \Sigma^*$, and a symbol $b \in \Sigma$, we say that the set $\{w \mid wb \in L\}$ is the *quotient language of* L *with respect to* b.

Theorem 8. **[Greibach Theorem]** Let us consider a class C of languages which is effectively closed under concatenation with regular sets and union. Let us assume that for that class C the problem of determining whether or not given a language $L \in C$, we have that $L = \Sigma^*$, for any sufficient large cardinality of Σ, is undecidable. Let P be a non-trivial property of C, that is, P is a non-empty subset of C different from C itself.

If P holds for all regular sets and it is preserved under quotient with respect to any symbol in Σ, then P is undecidable for C.

We have that:

- the class of context-free languages is effectively closed under concatenation with regular sets and union, and for context-free languages it is undecidable the problem of determining whether or not $L = \Sigma^*$ for $|\Sigma| \geq 2$,

- the class of regular languages is a non-trivial subset of the context-free languages,

- the property of being a regular language obviously holds for all regular languages, and

- the class of regular languages is closed under quotient with respect to any symbol in Σ (it is enough to delete the final symbol in the corresponding regular expression).

Thus, by Theorem 8, it is undecidable whether or not a given context-free grammar generates a regular language (see Property D5 on page 103).

Theorem 8 allows us to show that also *inherent ambiguity* for context-free languages is undecidable.

It is decidable whether or not given $k\,(\geq 0)$ and a context-free grammar G, G is $LR(k)$. This decidable problem is solvable in nondeterministic 1-exponential time [38, page 399] when k is given in binary, that is, it is solvable in $O(2^{p(n)})$ time by a nondeterministic Turing Machine, where $p(n)$ is a polynomial in the size of the input grammar G plus the size of k written in binary.

Given the context-free grammar G, it is decidable whether or not the language $L(G)$ is *infinite*. Indeed, it is enough: (i) to derive an equivalent grammar G_C in Chomsky normal form, without useless symbols, ε-productions, and unit productions, and (ii) to construct, as we now indicate, a directed graph D whose nodes are the nonterminals of G_C. In that graph D we construct an edge from node N_1 to node N_2 iff there exists a production in G_C such that $N_1 \to N_2\, A$ or $N_1 \to A\, N_2$ for some nonterminal A. We have that $L(G)$ is infinite iff in that graph there exists a cycle [16, page 137].

For context-free grammars (and, thus, for deterministic context-free grammars) it is decidable whether or not they generate the empty language. Indeed, it is enough to determine whether or not the axiom of the given grammar is a useful symbol.

Proposition 1. It does not exist an always terminating algorithm which given a context-free grammar G which is known to generate a regular language, constructs the regular grammar which is equivalent to G.

22.1 Decidable Properties of Deterministic Context-free Languages which are Undecidable for Context-free Languages

Let Σ be a given alphabet with at least two symbols. Now we list some *decidable* properties for deterministic context-free languages, subsets of Σ^*, which are *undecidable* for context-free languages, subsets of Σ^*, in the sense we specify below in Proposition 2 (see also [16, page 246]).

Given a *deterministic context-free grammar* G which generates the language $L(G)$, and given a regular language R, it is *decidable* whether or not:
(D1) $L(G) = R$,
(D2) $R \subseteq L(G)$,
(D3) $L(G) = \Sigma^*$, that is, the complement of $L(G)$ is empty,
(D4) $\Sigma^* - L(G)$ is a context-free language (recall that one can show that the complement of a deterministic context-free language is a deterministic context-free language),
(D5) $L(G)$ is a regular language, that is, it is decidable whether or not there exists a regular language $R1$ such that $L = R1$,
(D6) $L(G)$ is prefix-free, that is, it is decidable whether or not a deterministic context-free grammar G generates a language which is prefix-free [14, page 355].

Proposition 2. If G is known to be a *context-free grammar* (not a deterministic context-free grammar) then the problems (D1)–(D6) are all *undecidable.*

Recently the following result has been proved [36].
(D7) It is decidable whether or not $L(G1) = L(G2)$ for any two deterministic context-free grammars $G1$ and $G2$.

Recall that, on the contrary, it is undecidable whether or not $L(G1) = L(G2)$ for any two context-free grammars $G1$ and $G2$ (see Theorem 7 on page 101).

With reference to Property (D2), note that given a context-free grammar G and a regular language R, it is decidable whether or not $L(G) \subseteq R$. Indeed, we have that: (i) $L(G) \subseteq R$ iff $L(G) \cap (\Sigma^* - R) = \emptyset$, (ii) $L(G) \cap (\Sigma^* - R)$ is a context-free language, and (iii) emptiness of the language generated by a context-free grammar is decidable. The construction of the context-free grammar, say H, which generates the language $L(G) \cap (\Sigma^* - R)$, can be done in two steps: first, (i) we construct the pda M accepting $L(G) \cap (\Sigma^* - R)$ as indicated in [16, pages 135–136], and then, (ii) we construct H as the context-free grammar which is equivalent to M.

22.2 Undecidable Properties of Deterministic Context-free Languages which are Undecidable also for Context-free Languages

In this section we list some undecidable properties for deterministic context-free languages which are undecidable also for context-free languages in the sense we specify below in Proposition 3 [16, page 247].

Given any two *deterministic context-free grammars* $G1$ and $G2$, whose corresponding languages are $L1$ and $L2$, respectively, it is *undecidable* whether or not:

(U1) $L1 \cap L2 = \emptyset$,

(U2) $L1 \subseteq L2$,

(U3) $L1 \cap L2$ is a deterministic context-free language,

(U4) $L1 \cup L2$ is a deterministic context-free language,

(U5) $L1 \cdot L2$ is a deterministic context-free language, where $\cdot$ denotes concatenation of languages,

(U6) $L1^*$ is a deterministic context-free language,

(U7) $L1 \cap L2$ is a context-free language.

Proposition 3. If $G1$ and $G2$ are known to be *context-free grammars* (not deterministic context-free grammars) and in (U3)–(U6) we keep the word 'deterministic', then the problems (U1)–(U7) are still all *undecidable*.

22.3 Summary of Decidability and Undecidability Results in Formal Languages

In Figure 25 below we summarize some decidable and undecidable properties of the various kinds of grammars in Chomsky's Hierarchy. In this figure REG, DCF, CF, CS, and type 0 denote the classes of regular grammars, deterministic context-free grammars, context-free grammars, context-sensitive grammars, and type 0 grammars, respectively.

Now we list the Remarks (1)–(10) which clarify some entries of Figure 25.

(1) The problem '$L(G) = \Sigma^*$?' is trivial for the classes of grammars REG, DCF, CF, and CS, if we assume, as usual, that those grammars cannot generate the empty string ε. However, here we assume that the REG, DCF, and CF grammars are *extended grammars*, that is, they may have extra productions of the form $A \rightarrow \varepsilon$, and we also assume that CS grammars may have the extra production $S \rightarrow \varepsilon$ and the start symbol S does not occur in the right hand side of any production. With these hypotheses, the problem of checking whether or not $L(G) = \Sigma^*$, is not trivial and it is solvable or unsolvable as shown in the figure. Equivalently, we may assume that for the REG, DCF, CF, and CS class of grammars, the problem is, in fact, '$L(G) = \Sigma^+$?', rather than '$L(G) = \Sigma^*$?'.

(2) This problem can be solved by constructing the finite automaton which is equivalent to the given grammar.

(3) Having constructed the finite automaton, say F, corresponding to the given grammar G, we have that: (i) $L(G)$ is empty iff there are no final states in F, (ii) $L(G)$ is finite iff there are no loops in F.

problem \ grammar G	REG	DCF	CF	CS	type 0
is $w \in L(G)$?	S (2)	S	S	S	U (9)
is $L(G)$ empty ? finite ? infinite ?	S (3)	S	S	U	U (10)
is $L(G) = \Sigma^*$? (1)	S (4)	S	U (6)	U	U
is $L(G1) = L(G2)$?	S	S (5)	U (7)	U	U
is $L(G)$ context-free ?	S (yes)	S (yes)	S (yes)	U (8)	U
is $L(G)$ regular ?	S (yes)	S	U	U	U
is $L(G)$ inherently ambiguous ?	S (no)	S (no)	U	U	U
is G ambiguous ?	S (no)	S (no)	U	U	U

Fig. 25. Decidability of problems for various classes of languages and grammars. S means that the problem is solvable. The entry S (yes) means that the problem is solvable and the answer is always 'yes'. Analogously for the entry S (no). U means that the problem is unsolvable. The numbers (1)–(10) refer to the remarks we have made in the text.

(4) Having constructed the *minimal* finite automaton, say M, corresponding to the given grammar G, we have that $L(G)$ is equal to Σ^* iff M has one state only and for each symbol in Σ there is an arc from that state to itself.

(5) This problem has been shown to be solvable in [36]. Note that for DCF grammars the problem '$L(G1) \subseteq L(G2)$?' is unsolvable (see Property (U2) on page 104).

(6) It is undecidable whether or not given a context-free grammar G, we have that $L(G) = \Sigma^*$ (see Property (D3) on page 103).

(7) It is undecidable whether or not given two context-free grammars G_1 and G_2, we have that $L(G_1) = L(G_2)$ (see Section 22.2 and Theorem 7 on page 101).

(8) It is undecidable whether or not a context-sensitive grammar generates a context-free language [3, page 208].

(9) The membership problem for a type 0 grammar is a Σ_1-problem of the Arithmetical Hierarchy (see Section 24).

(10) The problem of deciding whether or not for a type 0 grammar(G), $L(G)$ is empty is a Π_1-problem of the Arithmetical Hierarchy (see Section 24).

23 Decidability and Undecidability of Tiling Problems

In this section we want to consider a different class of problems, the so called *tiling problems*, or *domino problems*. Actually, there are many different classes of tiling problems one might consider, and we will restrict our attention to a particular class which we now define.

An instance of a tiling problem is given by a *finite set of squares*, also called *tiles*. All squares have the same area. All tiles have colored edges, being the colors taken from a finite set of colors. The tiles differ only because of the different colors of their edges. Tiles have orientations and, thus, they *cannot* be rotated. Then we ask ourselves whether or not by using copies of the given tiles we can cover a given square area, either finite or infinite. In case the area is finite, we assume that its side is a multiple of the common size of the given tiles. When covering the given square area we have to comply with the constraint that two adjacent edges must have equal color.

Tiling problems can be formalized as follows.

Let N be the set of natural numbers. Let S be $N \times N$ or $\{0, \ldots, m\} \times \{0, \ldots, m\}$ for some $m > 0$. An instance of a tiling problem is given by a finite set D and two binary relations $H, V \subset D \times D$ (H stands for *horizontal*, and V stands for *vertical*). The triple $\mathcal{D} = \langle D, H, V\rangle$ is said to be a *tiling system*, or a *domino system*. Then we ask ourselves whether or not there exists a function $\tau : S \rightarrow D$, called a *tiling function*, such that:

$$\text{for all } x, y \in S, \quad \langle\tau(x, y),\, \tau(x{+}1, y)\rangle \in H \quad \text{and} \quad \langle\tau(x, y),\, \tau(x, y{+}1)\rangle \in V \qquad \text{(Adj)}$$

The set D is the set of tiles and the two binary relations H and V encode the matching of the colors of horizontally adjacent tiles and vertically adjacent tiles, respectively.

For instance, in Figure 26 we have depicted two tiles with colors taken from the set $\{a, b\}$. The colors a and b are encoded by a straight edge and an edge with a 'staple', respectively. By using those two tiles we can cover (and actually, in more than one way) any given square area of size $m \times n$, with $m, n \geq 1$. In Figure 26 we have shown a cover of a 5×5 area. It is easy to see that, indeed, the entire bidimensional plane can be covered by using the given two tiles.

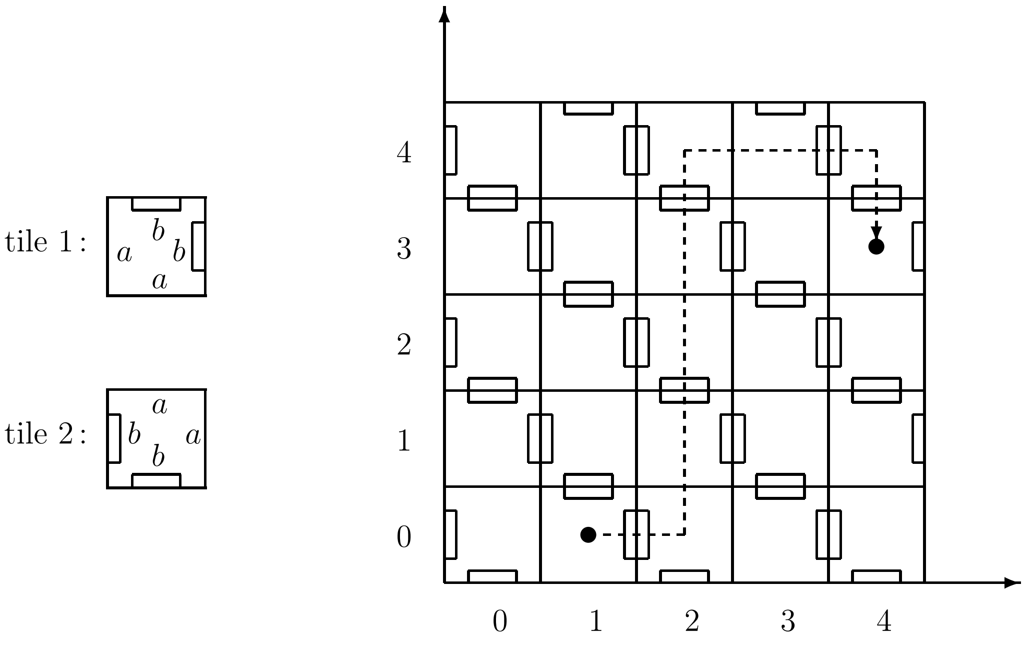


Fig. 26. Two tiles by which we can cover the entire plane. The colors a and b are encoded by a straight edge and an edge with a 'staple', respectively. The dashed line shows a 'snake' from $\langle 1, 0\rangle$ to $\langle 4, 3\rangle$.

If the given square area to be covered is finite, the problem is decidable simply because there is a finite number of possibilities to explore. We have that the tiling problem for square area of size $n \times n$, with $n \geq 1$, is NEXPTIME-complete if n is given in binary, and it is NP-complete if n is given in unary [29, page 501]. For these notions (which are complexity measures) the reader may refer to Sections 27 and 28.

We have that the *tiling problem is unsolvable* as stated by the following theorem.

Theorem 9. It does not exist an algorithm which for any given finite set of tiles, always provides an answer, and it answers 'yes' iff for all $m, n \geq 1$ one can cover the square area of size $m \times n$. Formally, it is *undecidable* whether or not given a tiling system $\mathcal{D} = \langle D, H, V \rangle$ there exists a tiling function $\tau : N \times N \rightarrow D$ satisfying the above constraints (Adj).

One can show that for any given finite set of tiles the following holds:
for all $m, n \geq 1$, every square of size $m \times n$ can be covered by using copies of the tiles *iff* the *entire* bidimensional plane can be covered by using copies of the tiles [13, page 189].

One can also show that the undecidability of the tiling problem implies that there exists a finite set of tiles such that by using copies of those tiles, we can cover the entire plane *aperiodically*, that is, *without translational symmetry*. However, in the cover of the entire plane there could be a rotational symmetry with respect to one point, or a mirror-image symmetry with respect to a symmetry line.

The absence of translational symmetry means that the pattern of tiles never repeats in an exact way. However, given a bounded region of the covered plane, no matter how large it is, that region will be repeated an infinite number of times within the cover of the entire plane.

It has been proved by R. Penrose that one can cover aperiodically the entire plane with infinite number of copies of two tiles only.

Now let us consider the following class of problems, called *domino snake problems*. An instance of a domino snake problem is given by a finite set of squares with colored edges of the kind we have considered in the tiling problems. We are also given two positions in the plane, an initial and a final position, called A and B, respectively. We have to place a *snake* of tiles so that the first tile is in A and the second tile is in B. (The notion of a snake will be formalized by that of a $\mathcal{D}$-thread below.) The possible positions for the snake is the entire plane, that it, the tiles of the snake can be placed anywhere in the plane. We have to comply with the only constraint that two adjacent tiles must have their edges of the same color and tiles cannot be rotated. In Figure 26 we have depicted with a dashed line the snake $[\langle 1,0\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 3,4\rangle, \langle 4,4\rangle, \langle 4,3\rangle]$ from the initial position $\langle 1,0\rangle$ to the final position $\langle 4,3\rangle$.

If the number of positions where the snake can be placed is finite, then the domino snake problem if decidable. We have the following two results (see, for instance, [13, Chapter 8]).

(i) If the snake can be placed everywhere in the entire bidimensional plane, then the domino snake problem is *decidable* (D. Myers proved in 1979 that it is PSPACE-complete [13, page 383]). For the notion of PSPACE-completeness the reader may refer to Section 28.

(ii) If the snake can be placed only in the upper bidimensional half-plane (say, for instance, the part above the horizontal line passing through the origin of the plane), then the domino snake problem is *undecidable* (H.-D. Ebbinghaus proved in 1982 that it is Σ_1-complete [13, page 383]. For the notion of Σ_1-completeness the reader may refer to the Arithmetical Hierarchy in Section 24).

Domino snake problems can be formalized as follows.

Let S_{mn} be the rectangle $\{0,\ldots,m\} \times \{0,\ldots,n\}$ for some $m, n > 0$. Let Θ be a *thread* in S_{mn}, that is, a finite sequence $[s_1 \ldots s_k]$ of elements of S_{mn} such that for all s_i, s_j with $i \neq j$ they have a common edge iff the two elements are adjacent in the sequence (that is, $|i-j| = 1$). The fact that the elements $\langle x_1, y_1\rangle$ and $\langle x_2, y_2\rangle$ in S_{mn} have a common edge can be formally expressed by the following property: $(x_1 = x_2 \wedge |y_1 - y_2| = 1) \vee (|x_1 - x_2| = 1 \vee y_1 = y_2)$.

Let $\mathcal{D} = \langle D, H, V\rangle$ be a tiling system. A $\mathcal{D}$*-thread* is a function $\tau : \Theta \rightarrow \mathcal{D}$ which covers the thread Θ by tiles in $\mathcal{D}$, that is, by elements of D which satisfy the constraints (Adj) imposed by H and V.

We have the following two results.

Theorem 10. (i) It is *decidable* whether or not, given a tiling system $\mathcal{D} = \langle D, H, V\rangle$ and a pair of numbers $\langle m, n\rangle \in N \times N$, there exists a $\mathcal{D}$-thread in $Z \times Z$ from $\langle 0, 0\rangle$ to $\langle m, n\rangle$.
(ii) It is *undecidable* whether or not, given a tiling system $\mathcal{D} = \langle D, H, V\rangle$ and a pair of numbers $\langle m, n\rangle \in N \times N$, there exists a $\mathcal{D}$-thread in $Z \times N$ (not $Z \times Z$) from $\langle 0, 0\rangle$ to $\langle m, n\rangle$.

Note that the undecidability of the domino snake problem for an infinite set A of possible positions for the snake, does *not* imply the undecidability of the domino snake problem for a set of possible positions which is an infinite proper subset of A. Thus, the undecidability of a problem does *not* depend on how large is the infinite search space which should be explored for finding a solution. In particular, a problem which is decidable and has an infinite search space A, can become undecidable if its search space is restricted to an infinite proper subset of A.

24 Arithmetical Hierarchy

Let us consider unary properties, that is, subsets, of the set N of natural numbers. We may also consider properties of arity $n > 1$, in which case they can be viewed as subsets of N^k. However, since there exists a bijection between N^k and N, we may consider any property of arity n, with $n \geq 1$, to be a subset of N.

Let us recall some notions we have introduced in Section 19. We say that a property (or a predicate) $R(x)$ of N is *decidable* (or *recursive*, or *solvable*) iff there exists a Turing Machine M that always halts and for any input value x in N, M tells us whether or not $R(x)$ holds by halting on a final or a non-final state, respectively.

This notion of decidability of a property is consistent with the notion of decidability of a problem. Indeed, the yes-language corresponding to a problem is a subset of Σ^* and that language can be viewed as a property which is enjoyed by some elements of Σ^*.

We say that a property $P(x)$ of N is *semidecidable* (or *semirecursive*, or *r.e.*, or *semisolvable*) iff there exists a Turing Machine M such that for any input value x in N for which $P(x)$ holds, we have that M enters a final state and halts.

Semidecidability is a particular 'degree of undecidability'. There are, indeed, various degrees of undecidability, and they can be structured in the so-called Arithmetical Hierarchy (see also Figure 27).

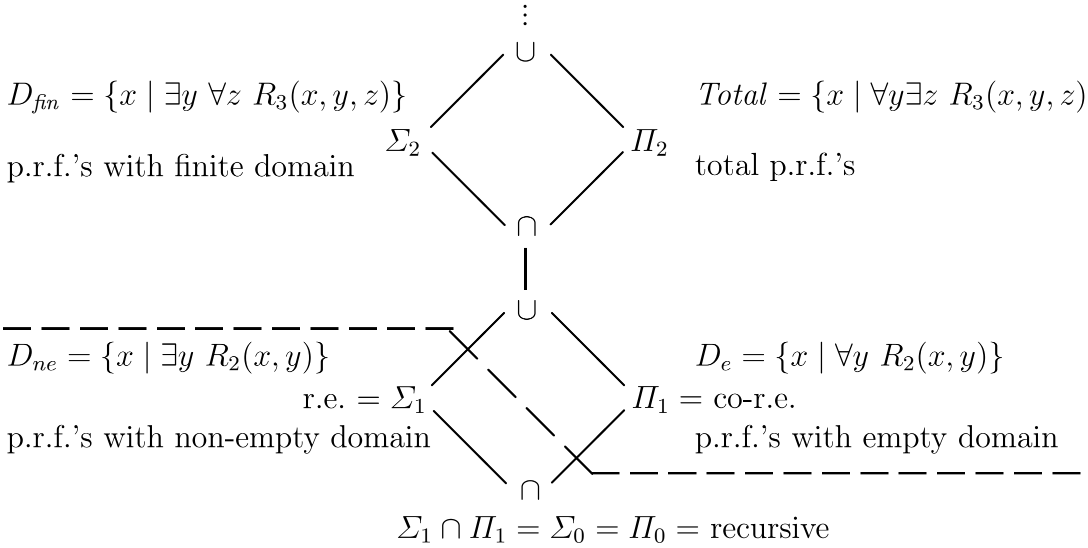

$D_{fin} = \{x \mid \exists y\ \forall z\ R_3(x,y,z)\}$
p.r.f.'s with finite domain
Σ_2
Π_2
$Total = \{x \mid \forall y \exists z\ R_3(x,y,z)\}$
total p.r.f.'s
$\cup$
$\cap$
$D_{ne} = \{x \mid \exists y\ R_2(x,y)\}$
r.e. = Σ_1
p.r.f.'s with non-empty domain
$D_e = \{x \mid \forall y\ R_2(x,y)\}$
Π_1 = co-r.e.
p.r.f.'s with empty domain
$\Sigma_1 \cap \Pi_1 = \Sigma_0 = \Pi_0$ = recursive

Fig. 27. The Arithmetical Hierarchy. R_2 and R_3 are recursive predicates. Turing computability is below the dashed line. p.r.f.'s stands for 'partial recursive functions'. Containments are all strict. A is contained in B iff A is drawn below B. $\cup$ is union and $\cap$ is intersection.

Definition 12. [Arithmetic Hierarchy] We say that the subset A of N is a Σ_n set (or simply Σ_n) iff for some $n \geq 0$, there exists a recursive predicate $R(x, y_1, \ldots, y_n)$ in N^{n+1}, such that

$A = \{x \mid \exists y_1 \forall y_2 \ldots\ R(x, y_1, \ldots, y_n)\}$. (All variables y_i's are alternatively quantified.)

We say that the subset A of N is a Π_n set (or simply Π_n) iff for some $n \geq 0$, there exists a recursive predicate $R(x, y_1, \ldots, y_n)$ in N^{n+1}, such that

$A = \{x \mid \forall y_1 \exists y_2 \ldots\ R(x, y_1, \ldots, y_n)\}$. (All variables y_i's are alternatively quantified.)

In order to understand the definition of the classes Σ_n and Π_n, the reader should also recall that a contiguous sequence of universal quantifications is equivalent to one universal quantification only. The same holds for existential quantifications. Thus, in the definitions of the classes Σ_n and Π_n, it is important both (i) the nature of the outermost quantification, and (ii) the number of alternations in the quantifications.

As a consequence of the above definitions we have that an r.e. subset of N is an element of Σ_1. Thus, the following is an alternative definition of an r.e. set.

A set A is said to be r.e. iff there exists a decidable predicate $R(x, y)$ such that $x \in A$ iff $\exists y\, R(x, y)$. The predicate R can be taken to be primitive recursive [6, page 124].

Given an r.e. set $\{x \mid P(x)\}$, the set $\{x \mid \mathit{not}(P(x))\}$ is said to be co-r.e. The set Π_1 is by definition the set of all co-r.e. subsets of N.

We have that $\Sigma_0 \cap \Pi_0$ is the set of the recursive subsets of N.

For any given $n \geq 1$, we have that [33, page 305] (see also Figure 27):

(i) $\{x \mid P(x)\}$ is in Σ_n iff $\{x \mid \mathit{not}(P(x))\}$ is in Π_n,

(ii) Σ_n and Π_n are both proper subsets of $\Sigma_n \cup \Pi_n$,

(iii) $\Sigma_n \cup \Pi_n$ is a proper subset of $\Sigma_{n+1} \cap \Pi_{n+1}$, and

(iv) $\Sigma_{n+1} \cap \Pi_{n+1}$ is a proper subset of both Σ_{n+1} and Π_{n+1}.

An example of a recursive predicate is the so-called *Turing Predicate* $T(i, x, y, t)$. $T(i, x, y, t)$ means that the Turing Machine whose encoding is the natural number i, given the input x, after exactly t steps, enters a final state and produces the output y.

Given a recursive property $R_2(x, y)$ we get a semirecursive property by considering its existential closure with respect to one of the two variables.

The set Dom_{ne} of partial recursive functions with non-empty domain, is in Σ_1, because

$\mathit{Dom}_{ne} = \{i \mid \exists x\, \exists y\, \exists t\; T(i, x, y, t)\}$.

The set Dom_{e} of partial recursive functions with empty domain, is in Π_1, because

$\mathit{Dom}_{e} = \{i \mid \forall x\, \forall y\, \forall t\; \mathit{not}(T(i, x, y, t))\}$.

The set Dom_{fin} of partial recursive functions with finite domain, is in Σ_2, because

$\mathit{Dom}_{fin} = \{i \mid \exists\, \mathit{max}\, \forall x\, \forall y\, \forall t\; \mathit{not}(T(i, x, y, t))\; \mathit{or}\; x < \mathit{max}\}$,

because:

$\exists\, \mathit{max}\, \forall x\;\; \mathit{if}\;\; (\exists y\, \exists t\; T(i, x, y, t))\; \mathit{then}\; x < \mathit{max}\quad \mathit{iff}$
$\exists\, \mathit{max}\, \forall x\, \forall y\, \forall t\; (\mathit{not}(T(i, x, y, t))\; \mathit{or}\; x < \mathit{max})$.

The set *Total* of partial recursive functions which are total, is in Π_2, because

$\mathit{Total} = \{i \mid \forall x\, \exists y\, \exists t\; T(i, x, y, t)\}$.

The set R_{Bound} of total partial recursive functions with finite image, is in $\Sigma_3 \cap \Pi_3$, because

$R_{Bound} = \{i \mid (\exists\, \mathit{max}\, \forall x\, \exists y\, \exists t\; T(i, x, y, t)\; \mathit{and}\; y < \mathit{max})\;\; \mathit{and}\;\; \forall x\, \exists y\, \exists t\; T(i, x, y, t)\}$.

As it is written, the predicate relative to the definition of R_{Bound}, indicates that R_{Bound} is in $\Sigma_3 \cap \Pi_2$, but since $\Pi_2 \subseteq \Pi_3$, it is also true that R_{Bound} is in $\Sigma_3 \cap \Pi_3$.

We have seen that the above sets Dom_e, Dom_{fin}, and *Total* are examples of Π_1, Σ_2, and Π_2 sets. They are, so to speak, significant examples, as we now illustrate via the following completeness notions.

Definition 13. [Hard and Complete Sets] Given two subsets, say A and B, of N, we say that A is 1-*reducible* to B, and we write$A \leq_1 B$, iff there exists an injective function f such that $\forall a \in A\;\, a \in A$ iff $f(a) \in B$.

Given a class C of subsets of N, and a reducibility relation $\leq$, we say that a set A is *hard with respect to* C *and* $\leq$ (or C-*hard with respect to* $\leq$, or $\leq$-*hard in* C) iff $\forall X \in C\;\, X \leq A$.

Given a class C of subsets of N, and a reducibility relation $\leq$, we say that a set A is *complete with respect to* C *and* $\leq$ (or C*-complete with respect to* $\leq$, or $\leq$*-complete in* C) iff $(\forall X \in C\ X \leq A)$ and $A \in C$.

We have that [40, page 43]:

(i) $K = \{i \mid \exists y\, \exists t.\, T(i, i, y, t)\}$ is Σ_1-complete with respect to $\leq_1$,

(ii) Dom_e is Π_1-complete with respect to $\leq_1$,

(iii) Dom_{fin} is Σ_2-complete with respect to $\leq_1$, and

(iv) $Total$ is Π_2-complete with respect to $\leq_1$.

The notion of hardness and completeness with respect to a class C of subsets of N and the reducibility relation $\leq_1$, can also be defined with respect to other classes of sets and other (reflexive and transitive) reducibility relations.

For instance, in the literature one finds the notions of NP-hard problems or NP-complete problems. These notions are given with respect to the class of NP problems and the logarithmic-space reductions [16, page 324]. (Actually, many authors use polynomial-time reductions rather than logarithmic-space reductions and we refer to [16] for more details on this matter.) We will consider these notions in the following chapter.

Before closing this section we would like to analyze the 'relative difficulty' of some problems in different classes of sets. Let us consider Figure 28 on page 112 and the three classes of sets indicated there:

1. REG (that is, the regular languages over $\Sigma = \{0, 1\}^*$),
2. CF (that is, the context-free languages over $\Sigma = \{0, 1\}^*$), and
3. PRF (that is, the partial recursive functions from N to N).

For each of these classes we consider the following three problems:

(1) the *Membership Problem*,

(2) the *Emptiness Problem*, and

(3) the *Totality Problem*.

We have that the relative difficulty of these problems is preserved within each class of sets, in the sense that we will specify in the following Points (α), (β), and (γ). In Figure 28 within each class of sets we have drawn 'more difficult' problems above 'easier' problems according to this layout:

Class of functions (or languages):

	(3) Totality Problem
(1) Membership Problem	(2) Emptiness Problem

Point (α). In the partial recursive functions the Membership Problem (3.1) is in Σ_1, while the Totality Problem (3.3) is 'more difficult' being in Π_2. The Emptiness Problem (3.2) is in Π_1.

Point (β). In the context-free languages the Membership Problem (2.1) is in Σ_0 (that is, it is decidable), while the Totality Problem (2.3) of determining whether or not a given

3. For $f \in$ PRF (partial recursive functions):

	(3.3) ‘is f total ?’ is in Π_2
(3.1) ‘is $x \in \mathrm{Dom}(f)$?’ is in Σ_1	(3.2) ‘is f totally undefined ?’ is in Π_1

2. For $L \in$ CF (context-free languages):

	(2.3) ‘is $L = \Sigma^*$?’ is in Π_1
(2.1) ‘is $w \in L$?’ is in Σ_0	(2.2) ‘is $L = \emptyset$?’ is in Σ_0

1. For $L \in$ REG (regular languages):

	(1.3) ‘is $L = \Sigma^*$?’ is PSPACE-complete
(1.1) ‘is $w \in L$?’ is in P	(1.2) ‘is $L = \emptyset$?’ is in P

Fig. 28. Comparison of problem difficulties in the classes of sets PRF, CF, and REG. Within each class more difficult problems are drawn above easier problems. The complexity classes: (i) P, that is, the polynomial time class, and (ii) PSPACE-complete, that is, the polynomial-space complete class (with respect to polynomial-time reductions) are defined with respect to the size of the regular expression denoting the regular language L (see also Sections 26–28).

language L is equal to Σ^*, is ‘more difficult’ being in Π_1. The Emptiness Problem (2.2) for context-free languages is in Σ_0.

Point (γ). In the regular languages the Membership Problem (1.1) can be solved in polynomial time (it is in P), while the Totality Problem (1.3) of determining whether or not a given regular language L is equal to Σ^*, is ‘more difficult’ being polynomial-space complete with respect to polynomial-time reductions. Indeed, as a consequence of Theorem 14 in Section 29.1 and Theorems 15 (ii.2) and 16 in Section 29.2, we will see that P $\subseteq$ PSPACE $\subseteq$ DTIME($2^{p(n)}$), where $p(n)$ is a polynomial in the size n of the regular expression denoting the language L. The Emptiness Problem (1.2) for regular languages is in P.

Remark 3. In the case of the regular languages we should have said that *maybe* the Totality Problem is ‘more difficult’ than the Membership Problem, because it is an open problem whether or not the containment P $\subseteq$ PSPACE is proper (see page 115).

More results on decidability and computability theory can be found in [8,15].

Chapter 4

Computational Complexity

25 Preliminary Definitions

We start off by indicating the models we will consider for measuring the complexity of computations.

In order to measure the space complexity, we will consider the off-line k-tape deterministic Turing Machine model with one read-only input tape, and $k\,(\geq 1)$ semi-infinite *storage tapes* (also called *working tapes*). We will use this model because the off-line Turing Machine allows us to consider space complexities less than linear.

By $S(n)$ we will denote the space complexity of a computation relative to an input of size n, that is, the maximum number of tape cells used in any of the k storage tapes. In order to capture the intuition that at least one memory cell is used in any computation, by $S(n)$ we actually mean the integer not smaller than $max(1, S(n))$, for any $n \geq 0$.

We denote by DSPACE($S(n)$) the family of languages (or problems) recognized by a *deterministic* Turing Machine by a computation whose space complexity is $S(n)$.

We denote by NSPACE($S(n)$) the family of languages (or problems) recognized by a *nondeterministic* Turing Machine by a computation whose space complexity is $S(n)$.

If $S(n) \geq n$ for any input of size n, we may restrict k to be 1, without changing the class DSPACE($S(n)$). The same holds for the class NSPACE($S(n)$).

For measuring the time complexity of a computation, we will consider the two-way infinite k-tape deterministic Turing Machine model. When studying the time complexity, one should, in general, specify the number k of tapes of the Turing Machine considered (see Theorem 4 below).

By $T(n)$ we will denote the time complexity of a computation relative to an input of size n, that is, the number of moves which the Turing Machine makes before halting.

In order to capture the intuition that in any computation the whole input of size n should be read, by $T(n)$ we actually mean the integer not smaller than $max(n+1, T(n))$ for any $n \geq 0$. We have $max(n+1, T(n))$, instead of $max(n, T(n))$, because one time unit is required to check that the input is terminated.

Note that there are Turing Machines which compute their output without reading their input (they are Turing Machines which compute constant functions) and in our theory of computational complexity we will *not* consider them.

We denote by DTIME($T(n)$) the family of languages (or problems) recognized by a *deterministic* Turing Machine by a computation whose time complexity is $T(n)$.

We denote by NTIME($T(n)$) the family of languages (or problems) recognized by a *nondeterministic* Turing Machine by a computation whose time complexity is $T(n)$.

Example 1. Let us consider the language $L = \{w\, w^R \mid w \in \{0,1\}^*\}$, where w^R denotes 'w reversed', that is, $\varepsilon^R = \varepsilon$, $(w\,0)^R = 0\,w^R$, and $(w\,1)^R = 1\,w^R$. L can be recognized in time complexity $2n+1$, where n is the length of w. L can also be recognized in $O(\log_2 n)$ space complexity as the following algorithm indicates:

```
let $n$ be the length of the word $w$;
for $i = 1, \ldots, |w|$, let $w_i$ be the $i$-th symbol (from the left) of $w$;
for $i = 1$ to $n$ check whether or not $(w\,w^R)_i = (w\,w^R)_{n-i+1}$
    (this check can be done by counting symbols and testing for equality)
```

Since this algorithm uses numbers (written in binary) and tests for 0 and 1 only, it requires only $O(\log_2 n)$ cells on the off-line Turing Machine storage tape. □

Theorem 1. [Tape Compression] If a language L is accepted by an off-line k-tape deterministic Turing Machine in space $S(n)$ then for any $c > 0$, the language L is accepted also by an off-line k-tape deterministic Turing Machine in space $c\,S(n)$.

Theorem 2. [From k tapes to 1 tape with respect to space complexity] If a language L is a language accepted by an off-line k-tape deterministic Turing Machine in space $S(n)$, then L is accepted also by an off-line 1-tape deterministic Turing Machine in space $S(n)$.

The proof of this theorem is based on the use of tracks on one tape to simulate k tapes.

From now on, unless otherwise specified, for measuring space complexity we will refer to a Turing Machine M such that: (i) M has the input tape and only one storage tape, and (ii) if $S(n) \geq n$ then M has exactly one storage tape and not the input tape (that is, the input is assumed to be written in the leftmost part of the storage tape).

Given a function $f(n)$, let $sup_{n\rightarrow+\infty} f(n)$ denote the least upper bound of $f(n)$ for $n \rightarrow +\infty$, and let $inf_{n\rightarrow+\infty} f(n)$ denote the greatest lower bound of $f(n)$ for $n \rightarrow +\infty$.

Theorem 3. [Linear Speed-up] If a language L is accepted by a k-tape deterministic Turing Machine in time $T(n)$ with $k>1$ and $inf_{n\rightarrow+\infty} \frac{T(n)}{n} = +\infty$ (that is, $T(n)$ is growing more than linearly), then for any $c > 0$, L is accepted by a k-tape deterministic Turing Machine in time $c\,T(n)$.

For any $k > 1$, if L is a language accepted by a k-tape deterministic Turing Machine in time $T(n)$ and $T(n) = c\,n$ for some $c>0$ (that is, $T(n)$ is linear), then for any $\varepsilon>0$, L is accepted by a k-tape deterministic Turing Machine in time $(1+\varepsilon)\,n$.

Theorem 4. [Reduction of tapes] For any $k > 1$, if L is a language accepted by a k-tape deterministic Turing Machine in time $T(n)$, then L is accepted by a two-tape deterministic Turing Machine in time $T(n)\ \log(T(n))$ and L is accepted by a 1-tape deterministic Turing Machine in time $T^2(n)$.

Theorems 1, 2, 3, and 4 hold also when replacing 'deterministic Turing Machine' by 'nondeterministic Turing Machine'.

Recall that we are not interested in the base of the logarithms, because, as indicated by Theorems 1 and 3, multiplicative constants are not significant for the validity of most of the complexity results we will present.

We will use the following abbreviations:

$$\begin{aligned}
\text{EXPSPACE} &=_{def} \textstyle\bigcup_{k>0} \text{DSPACE}(2^{n^k})\\
\text{NEXPTIME} &=_{def} \textstyle\bigcup_{k>0} \text{NTIME}(2^{n^k})\\
\text{EXPTIME} &=_{def} \textstyle\bigcup_{k>0} \text{DTIME}(2^{n^k})\\
\text{PSPACE} &=_{def} \textstyle\bigcup_{k>0} \text{DSPACE}(n^k) \overset{(1)}{=} \text{NSPACE} =_{def} \textstyle\bigcup_{k>0} \text{NSPACE}(n^k)\\
\text{NP} &=_{def} \textstyle\bigcup_{k>0} \text{NTIME}(n^k)\\
\text{P} &=_{def} \textstyle\bigcup_{k>0} \text{DTIME}(n^k)
\end{aligned}$$

We have that:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \overset{(1)}{=} \text{NSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE}.$$

The equality $\overset{(1)}{=}$ above, that is, PSPACE = NSPACE, is established in Theorem 16 on page 143.

It is an open problem
(i) whether or not P = NP (see page 124 at the end of Section 26),
(ii) whether or not NP = PSPACE [19, page 98],
(iii) whether or not PSPACE = EXPTIME [19, page 102], and
(iv) whether or not EXPTIME = NEXPTIME [19, page 103].

Note that it is also an open problem whether or not P = PSPACE. If P = NP then EXPTIME = NEXPTIME [19, page 103]. We will return to these issues in Section 28.6.

26 P and NP Problems

Let us begin this section by giving the definition of the classes P and NP of problems (or languages, or predicates). Actually, we will give three equivalent definitions of the class NP.

Informally, we may say that the class P of problems (or languages, or predicates) corresponds to the class of predicates which can be evaluated in polynomial time with respect to the size of the input by a deterministic algorithm. More formally, we have the following definition.

Definition 1. [**Class** P. **Polynomial Class**] A predicate $p(x)$ is in the class P iff there exists a deterministic Turing Machine M and a polynomial $r(n)$ such that for any input x of size n, M evaluates $p(x)$ in a sequence of moves whose length is at most $r(n)$.

Note that in the polynomial $r(n)$ we do not care about the exponent of the highest power of n. Thus, here and in the sequel, by the phrase “in polynomial time with respect to n” where n is the size of the input, we mean “in a sequence of moves whose length is at most $c\,n^k$, for some constants $c>0$ and $k\geq 1$, independent of n”.

A definition of the class of NP problems (or languages, or predicates) can be given in terms of nondeterministic Turing Machines which at each step of the computation have the choice of one among a bounded number of possibilities, and that bounded number is independent of the input value. Recall that an input word is accepted by a Turing Machine M iff *at least one* of the possible sequences of moves of M leads from the initial state to a final state.

Definition 2. [**Class** NP. **Version 1**] A predicate $p\colon D \to \{\mathit{true}, \mathit{false}\}$ is in the class NP iff there exists a nondeterministic Turing Machine M such that for all $d \in D$,
(i) if $p(d)$ holds then M evaluates $p(d)$ to *true* in polynomial time with respect to the size of the input d, and
(ii) if $p(d)$ does *not* hold then M evaluates $p(d)$ to *false* in finite time, but nothing is said about how much time it takes in this case.

Now we will give two more definitions of the class NP, but we will not provide the proof of their equivalence. The first definition captures the intuition that the class NP of problems is the class of predicates which can be evaluated in polynomial time by a deterministic Turing Machine after 'a search of polynomial depth'.

Definition 3. [**Class** NP. **Version 2**] A predicate $p\colon \to \{\mathit{true}, \mathit{false}\}$ is in the class NP iff it can be evaluated as follows:

(i) there exists a finite alphabet Σ,

(ii) there exists a predicate $\pi\colon D \times \Sigma^* \to \{\mathit{true}, \mathit{false}\}$ which for every $d \in D$ and $w \in \Sigma^*$, is evaluated by a deterministic Turing Machine in polynomial time with respect to $\mathit{size}(d) \times \mathit{size}(w)$, and

(iii) there exists $k \geq 0$ such that for all $d \in D$,
Step (1): we consider the set of words $W = \{w \mid w \in \Sigma^* \text{ and } \mathit{size}(w) = (\mathit{size}(d))^k\}$, and
Step (2): we return the truth value of $p(d) = \text{if } \exists w \in W\ \pi(d, w) \text{ then } \mathit{true} \text{ else } \mathit{false}$.

Since for any $d \in D$, the predicate π can be evaluated in polynomial time with respect to $\mathit{size}(d) \times \mathit{size}(w)$, that is, $(\mathit{size}(d))^{k+1}$, we have that π can be evaluated in polynomial time with respect to $\mathit{size}(d)$.

According to Definition 2, the evaluation of the predicate p can be viewed as a restricted type of exponentially long computation, in the sense that we may evaluate p by constructing in a nondeterministic way, a tree of polynomial depth using a set W of words, and then computing $\pi(d, w)$ for the word w at each leaf. These computations at the leaves can be carried out independently, in parallel. Only at the end, one should synchronize their results for obtaining the required value, that is, $\exists w \in W\ \pi(d, w)$ (see Figure 29 on page 117). Thus, the evaluation of the predicate p can also be viewed as the search in a polynomially deep tree for a leaf, if any, associated with a word w, such that $\pi(d, w) = \mathit{true}$.

Here is a third definition of the class NP.

Definition 4. [**Class** NP. **Version 3**] A predicate $p\colon D \to \{\mathit{true}, \mathit{false}\}$ is in the class NP iff
(i) there exists a finite alphabet Σ,
(ii) there exists a function $c : D \to \Sigma^*$ such that
- for any $d \in D$, the string $c(d)$, called the *certificate* of d, is in Σ^*, and
- $c(d)$ is evaluated by a deterministic Turing Machine in polynomial time with respect to $\mathit{size}(d)$, and

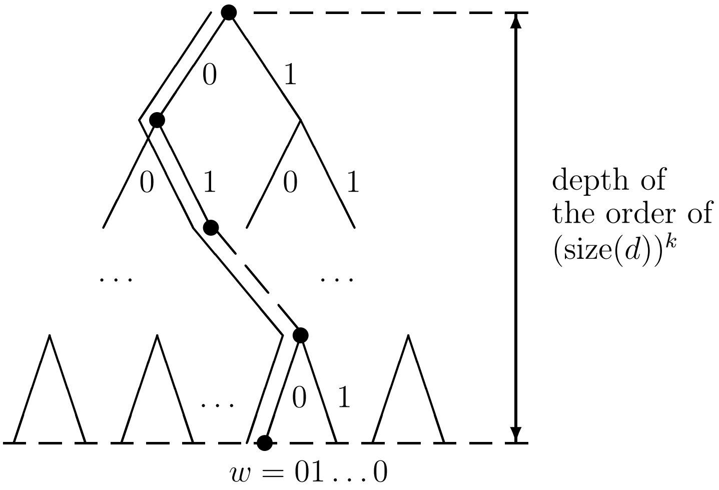


Fig. 29. The nondeterministic construction of a polynomially deep tree. We have assumed that $\Sigma = \{0,1\}$. $d \in D$ is the input value whose size is denoted by *size*(d).

(iii) there exists a function $\alpha : D \times \Sigma^* \rightarrow \{true, false\}$ which
- takes an element $d \in D$ and the corresponding string $c(d) \in \Sigma^*$ and
- can be evaluated by a deterministic Turing Machine in polynomial time with respect to *size*(d) × *size*$(c(d))$ such that

for all $d \in D$, $p(d) = true$ iff $\alpha(d, c(d)) = true$.

Note that: (i) by our hypotheses, the function α can be computed in polynomial time with respect to *size*(d), because *size*$(c(d))$ is polynomial with respect to *size*(d) (because $c(d)$ is computed in polynomial time w.r.t. d), and (ii) with reference to Definition 4, if $p(d)$ is *true*, then the certificate d can be identified with a word w for which $\pi(d, w)$ is *true*.

In the above definitions of the classes P and NP the reference to the Turing Machine is not actually necessary: indeed, one may refer also to other model of computations (see Section 31 below), and thus, we may replace the concept of 'a Turing Machine' by that of 'an algorithm'.

We now show that some problems are in NP. Let us first consider the following problem.

MINIMAL SPANNING TREE

Input: a weighted undirected connected graph G with N nodes and a weight k. Weights are assumed to be natural numbers.

Output: 'yes' iff G has a spanning tree of weight *not* larger than k.

This problem (which is a variant of the one described in [32, page 101]) is in NP because it can be solved in the following two steps. Let us consider Σ to be the set $\{0,1,2\}$.
Step (1): we construct the set $W \subseteq \Sigma^*$ of the encodings of all $(N-1)$-tuples of arcs of G (recall that the spanning tree of a graph with N nodes has $N-1$ arcs), and

Step (2): we return 'yes' iff there exists a $(N\!-\!1)$-tuple $w \in W$ such that w represents a spanning tree, that is, it includes all nodes of G, and the sum of the weights of w is not larger than k.

These two steps can be described in more details as follows.

Step (1). Let us assume that every node is denoted by a natural number n, with $0 \leq n \leq N\!-\!1$. Then any arc $\langle m, n \rangle$ of G can be represented by a sequence of the form:

$\alpha\, 2\, \beta\, 2\, \gamma$, called the encoding of the arc $\langle m, n \rangle$,

where α, β, and γ are sequences in $\{0, 1\}^+$ which represent the binary expansion of m, the binary expansion of n, and the weight of the arc $\langle m, n \rangle$, respectively. The graph G can be encoded by the sequence of the form:

(22 'encoding of an arc')$^+$ 22.

Let h be the maximum weight on the arcs of G. Let A_m be the length of the longest encoding of an arc. Obviously, A_m is a polynomial function of N and h.

Thus, Step (1) consists in constructing the set W of *all strings* of the form:

$(22\, \alpha\, 2\, \beta\, 2\, \gamma)^{N-1}\, 22$,

where the length of $\alpha\, 2\, \beta\, 2\, \gamma$ is not greater than A_m.

In the set W, besides other strings, there are the encodings of all spanning trees of G. Actually, every spanning tree is, in general, represented in W more than once, because the order in which the arcs of the tree are listed is not relevant. Some strings in W may be ill-formed or may not represent a tree (this is the case, for instance, when an arc is represented more than once).

The length of a word in Σ^* which encodes a $(N\!-\!1)$-tuple of arcs of G, is of the order of $(N\!-\!1) \times A_m$ (+ lower order terms), and thus, it is bounded by a polynomial of N and h.

Step (2) consists in testing for each string w in W whether or not w represents a spanning tree of G with total weight not larger than k. For each string w that test can be done in polynomial time, because it is enough to scan the string w and check whether or not

- it represents a subgraph of G which includes all nodes of G, and
- it does not have cycles, and
- the total weight of its arcs is not larger than k.

Note that a subgraph of G is a tree iff it does not have cycles, and the presence of a cycle in a given set of arcs can be detected by matrix multiplication (see, for instance, [32, Chapter 1]).

Actually, the minimal spanning tree problem is in P. Indeed, given any weighted undirected graph G, using Kruskal's algorithm we may compute a spanning tree of minimal weight in polynomial time (see, for instance, [32, page 101]).

Let us consider the following problem on undirected graphs.

HAMILTONIAN CIRCUIT
Input: a weighted undirected connected graph G with N nodes.

Output: 'yes' iff in G there exists a node n and a path from n to n (that is, a cycle including the node n) which visits every node of G exactly once.

This problem is in NP because it can be solved as indicated by the following steps.
Step (1): we construct the set W of the encodings of all $(N-1)$-tuples of nodes of G, and
Step (2): we return 'yes' iff in W there exists an encoding w of an $(N-1)$-tuple of nodes which is a cycle which visits every node of G exactly once.

Step (1) can be performed as indicated in the case of the minimal spanning tree problem. With reference to Step (2) it is not difficult to see that the test whether or not a given tuple w represents a cycle which visits exactly once every node of G can be performed in polynomial time with respect to the size of the encoding of the input graph G.

Let us consider the following problem.

CO-PRIME
Input: a positive integer q in binary notation, using $k = \lceil \log_2 q \rceil$ bits.

Output: 'yes' iff q is *not* a prime.

The 'CO-' prefix in the name of this problem stands for 'complement of', in the sense that the output is 'yes' iff the input is *not* a prime.

Proposition 1. The CO-PRIME problem is in NP.

Indeed, an instance of the CO-PRIME problem can be solved as indicated by the following two steps.

Step (1): we construct the set $W \in \{0,1\}^k$ of the encodings of all positive numbers i, for $1 < i < q$, and

Step (2): we return 'yes' iff there exists an encoding $w \in W$ such that w represents a number which divides q. Obviously, to test whether or not a given number divides q requires polynomial time with respect to k.

Now let us consider the following argument. Two factors of a given number p can be chosen in p^2 ways. Each pair can be multiplied in $O(p^2)$ time. So in $O(p^4)$ time we may know whether or not a number is non-prime.

However, by this argument we *cannot* conclude that CO-PRIME is in NP, because the size of the input is $\lceil \log_2 p \rceil$ (and not p), and since $p \leq 2^{\lceil \log_2 p \rceil}$, we have that it is possible to solve the CO-PRIME problem in $O(2^{4\lceil \log_2 p \rceil})$ time, which is an exponential amount of time with respect to the size of the input.

Let us also consider the complement problem of CO-PRIME, called PRIME, defined as follows.

PRIME

Input: a positive integer p in binary notation.

Output: 'yes' iff p is a prime.

In [1] the reader may find a proof that PRIME is in P. Thus, by that proof we may conclude that also CO-PRIME is in P.

We have the following proposition.

Proposition 2. The problem PRIME is in NP.

The reader should realize that the complement (or negation) of a problem in NP is *not*, in general, in NP. Indeed, in the complement problem, that is, the CO-PRIME problem, the existential quantifier of Step (2) becomes a universal quantifier. Thus, having shown above that CO-PRIME is in NP, we cannot conclude that PRIME is in NP.

Let us begin the proof that PRIME is in NP by noticing that the size of the input to the problem is $\lceil \log_2 p \rceil$, for any given input p. In the proof we will write $a = b \pmod p$ to denote that $\mid a - b \mid = k\,p$ for some integer number $k \geq 0$. We use the following theorem.

Theorem 5. **[Fermat's Theorem]** A number $p\,(> 2)$ is prime iff there exists x with $1 < x < p$, such that
Condition (α): $x^{p-1} = 1 \pmod p$ and
Condition (β): for all i, with $1 \leq i < p{-}1$, $x^i \neq 1 \pmod p$.

Note that since for x such that $1 < x < p$, we have that: $x^1 \neq 1 \pmod p$, and thus, in Condition (β) of the above Theorem 5 we can write $1 < i < p{-}1$, instead of $1 \leq i < p{-}1$.

Example 2. Let us consider the prime number $p = 7$. We have that there exists the number $x = 3$ such that: $1 < x < 7$ and
Condition (α) holds because: $3^{7-1} = 729 = 1 \pmod 7$, and
Condition (β) holds because:

$3^1 = 3 \pmod 7$
$3^2 = 2 \pmod 7$ (indeed, $3 \times 3 = 9 = 7{+}2$)
$3^3 = 6 \pmod 7$ (indeed, $2 \times 3 = 6$)
$3^4 = 4 \pmod 7$ (indeed, $6 \times 3 = 18 = (2 \times 7){+}4$)
$3^5 = 5 \pmod 7$ (indeed, $4 \times 3 = 12 = 7{+}5$) □

Given two positive integers n and m, by $rem(n/m)$ we denote the remainder of the division of n by m. Given an integer number $m\,(\geq 2)$ we define *pfactors*(m) (where the name '*pfactors*' stands for '*prime factors*') to be the set $\{p_j \mid p_j \geq 2$ and p_j is prime and $rem(m/p_j) = 0\}$.

For any $m \geq 2$, the cardinality of the set *pfactors*(m) is at most $\lfloor \log_2 m \rfloor$. For instance, we have that *pfactors*$(2) = \{2\}$ whose cardinality is $1 = \lfloor \log_2 2 \rfloor$.

Let us first recall the following theorem.

Theorem 6. For $1 < x, y < p$:
(1) $x + y \pmod p$ takes $O(\log_2 p)$ time,
(2) $xy \pmod p$ takes $O((\log_2 p)^2)$ time,
(3) the computation of the quotient and the remainder of the integer division $(\bmod\ p)$ of x by y takes $O((\log_2 p)^2)$ time, and
(4) $x^y \pmod p$ takes $O((\log_2 p)^3)$ time.

Proof. (1), (2), and (3) are immediate because when computing in the modular arithmetics modulo p, we need at most $\lceil \log_2 p \rceil$ bits. (4) derives from the fact that we can compute $x^y \pmod p$ by computing $x \pmod p$, $x^2 \pmod p$, $x^4 \pmod p$, ... (this takes at most $\log_2 p$ multiplications because y is smaller than p) and then multiplying the powers of x which correspond to the 1's of the binary expansion of y (and this takes at most $\log_2 p$ multiplication). □

For testing Condition (α) we construct the binary encoding of every x, with $1 < x < p$, as a path of a binary tree. The length of each of these encodings is at most $\lceil \log_2 p \rceil$. Thus, the test of Condition (α) is bounded by a polynomial of the length $\log_2 p$ of the input (see Figure 30).

For any such x we have to test whether or not $x^{p-1} = 1 \pmod p$ and by Theorem 6, each test can be done in $O((\log_2 p)^3)$ time.

For each x, with $1 < x < p$, we have also to test Condition (β), that is, we have to test whether or not for all i, $1 < i < p{-}1$, $x^i \neq 1 \pmod p$. (Recall that the test for $i = 1$ is not necessary because we have that $x \neq 1$.) These tests are 'too many' to show that PRIME is in NP, because $p{-}1$ is of the order of $2^{\log_2 p}$ which is an exponential with respect to the size of the input, which is $\log_2 p$. Thus, the total time is not within polynomial time with respect to $\log_2 p$. However, the number of these tests can be reduced by using the result of Theorem 9 below, which we will state and prove below.

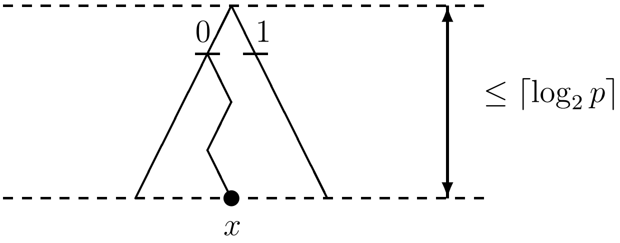


Fig. 30. Finding the binary encoding of every x, with $1{<}x{<}p$.

Let us first recall that: $a \pmod p + b \pmod p = (a + b) \pmod p$, and
$a \pmod p \times b \pmod p = (a \times b) \pmod p$.

To show Theorem 9 we first need the following Theorems 7 and 8.

Theorem 7. If $x^i = 1 \pmod p$ for some i then $x^{ki} = 1 \pmod p$ for any integer number $k \geq 1$.

Proof. Any power of 1 is 1. □

Theorem 8. Let us assume that $p > 2$ and $1 < x < p$.
If (Condition A) $x^{p-1} = 1 \pmod{p}$ and
(Condition B) i is equal to the *smallest* natural number h such that $1 < h < p-1$ and $x^h = 1 \pmod{p}$ then
(Condition C) i divides $p-1$.

Proof. Let us consider the following Condition $B^{\#}$:

i is a natural number such that $1 < i < p-1$ and $x^i = 1 \pmod{p}$ $\qquad (B^{\#})$

We have that: (B) implies $(B^{\#})$. To show that: if (A) and (B) then (C), it is enough to show that: if (A) and (B) and $(B^{\#})$ then (C), that is, if (not C) and (A) and $(B^{\#})$ then (not B).

By (not C), if i does not divides $p-1$ we have that: $p-1 = k\,i+r$, for $0 < r < i$. Thus, $x^{p-1} \pmod{p} = x^{ki+r} \pmod{p} = x^r \pmod{p}$, by $(B^{\#})$ and Theorem 7. By (A) we have that $x^{p-1} = 1 \pmod{p}$ and thus, we also have that $x^r \pmod{p} = 1$. Moreover, we have that $r \neq 1$, because $1 < x < p$ and $x^r \pmod{p} = 1$. Thus, $1 < r < i$. Hence, i is not equal to the smallest natural number h such that $1 < h < p-1$ and $x^h = 1 \pmod{p}$. □

Theorem 9. Let us assume that: (i) $p>2$, (ii) $1<x<p$, and (iii) $x^{p-1}=1 \pmod{p}$.
If for all $p_j \in \mathit{pfactors}(p-1)$, $x^{(p-1)/p_j} \neq 1 \pmod{p}$ then for all i, with $1 < i < p-1$, we have that $x^i \neq 1 \pmod{p}$.

Proof. Let us prove the contrapositive. Let us assume that there exists i, with $1<i<p-1$, such that $x^i = 1 \pmod{p}$. Then, there exists a smallest such i and, by Theorem 8, we may assume that i divides $p-1$. That is, $i = (p-1)/(b_1 \times \ldots \times b_r)$ for some (not necessarily distinct) $b_1, \ldots, b_r$ in $\mathit{pfactors}(p-1)$ and $r > 0$.

Thus, $x^{(p-1)/(b_1 \times \ldots \times b_r)} = 1 \pmod{p}$ for some $b_1, \ldots, b_r$ in $\mathit{pfactors}(p-1)$ and $r > 0$. By Theorem 7, for any natural number $q \geq 1$, $x^{q(p-1)/(b_1 \times \ldots \times b_r)} = 1 \pmod{p}$. Thus, by choosing $q = b_2 \times \ldots \times b_r$ if $r \geq 2$, or $q = 1$ if $r = 1$, we get that there exists p_j in $\mathit{pfactors}(p-1)$ such that $x^{(p-1)/p_j} = 1 \pmod{p}$. □

Thus, if $1 < x < p$ and $x^{p-1} = 1 \pmod{p}$ then, in order to test that

for all i, with $1 < i < p-1$, $x^i \neq 1 \pmod{p}$

it is enough to test that

for all $p_j \in \mathit{pfactors}(p-1)$, $x^{(p-1)/p_j} \neq 1 \pmod{p}$.

These tests are *not* 'too many', because the cardinality of $\mathit{pfactors}(p-1)$ is at most $\lfloor \log_2 p \rfloor$.

Example 3. With reference to Example 2 on page 120, in order to test Condition (β) we need to test that:

$3^1 \neq 1 \pmod{7}$
$3^2 \neq 1 \pmod{7} \qquad (*)$
$3^3 \neq 1 \pmod{7} \qquad (*)$
$3^4 \neq 1 \pmod{7}$
$3^5 \neq 1 \pmod{7}$.

Indeed, all these inequalities hold, because

$3^1 = 3 \pmod{7}$,
$3^2 = 9 = 2 \pmod{7}$,

$3^3 = 27 = 6 \pmod 7$,
$3^4 = 81 = 4 \pmod 7$, and
$3^5 = 243 = 5 \pmod 7$.

Now Theorem 9 tells us that in order to test Condition (β), it is enough to test that $3^{6/2} \neq 1 \pmod 7$ and $3^{6/3} \neq 1 \pmod 7$, because *pfactors*$(6) = \{2, 3\}$. Thus, we have to test only the above disequations marked with $(*)$. □

As a consequence of Theorem 9, in order to test Condition (β) we first construct all prime factorizations $p_1 \times \ldots \times p_k$ of $p-1$, that is, we construct all lists $[p_1, \ldots, p_k]$ such that:
(a) $p_1 \times \ldots \times p_k = p-1$ and
(b) for $j = 1, \ldots, k$, we have that: p_j is a prime number and $2 \leq p_j \leq p-1$.

For instance, we have the following three prime factorizations of 12: $[2, 2, 3]$, $[2, 3, 2]$, and $[3, 2, 2]$.

The prime factorizations of a number, say m, can be constructed in two steps as follows.
Step (1): we first generate all possible lists of length at most $\lfloor \log_2 m \rfloor$ of (not necessarily distinct, not necessarily prime) numbers each of which is in the interval $[2, \ldots, m]$ (and, thus, each number can be represented with at most $\lceil \log_2 m \rceil$ bits), and then
Step (2): we select among those lists, those of the form $[m_1, \ldots, m_k]$ such that: for $j = 1, \ldots, k$, we have that m_j is prime, and $m_1 \times \ldots \times m_k = m$.

The lists of Step (1), called *candidate factorizations*, can be viewed as words in $\{0, 1\}^*$, each of which consists of at most $\lfloor \log_2 m \rfloor$ subwords and each subword is at most of $\lceil \log_2 m \rceil$ bits. These words can be arranged as paths from the root to the leaves of a binary tree (see Figure 31.

Note that that binary tree is polynomially deep with respect to the size of the input which is $\lceil \log_2 p \rceil$. Its depth is at most $\lfloor \log_2 (p-1) \rfloor \times \lceil \log_2 p \rceil$.
Now we will measure the cost of testing Condition (β), after constructing the tree of the candidate factorizations of $p-1$.

Let us assume that we can solve the PRIME problem in $c \lceil \log_2 p \rceil^4$ time, for some constant $c > 0$, for any number p whose size is $\lceil \log_2 p \rceil$. (From now on what follows we will feel free to write $\log_2 p$, instead of $\lceil \log_2 p \rceil$ or $\lfloor \log_2 p \rfloor$, because by doing so we do not modify the asymptotic complexity measures.)

We have the following three contributions ($\beta 1$), ($\beta 2$), and ($\beta 3$) to the cost of testing Condition (β).

($\beta 1$) We check that $p_1, \ldots, p_k$ are all prime in time $\Sigma_{1 \leq j \leq k}\, c\, (\log_2 p_j)^4$ (by induction hypothesis).

($\beta 2$) We check that $p-1 = p_1 \times \ldots \times p_k$ by performing $(k-1)$ $(\leq \log_2 p)$ multiplications each of $\log_2 p$ bits. (Recall that the multiplications are modulo p. So we need at most one extra bit to test whether or not the result of a multiplication is larger than $p-1$.) These multiplications take $c_1 (\log_2 p)^3$ time, for some constant $c_1 > 0$.

($\beta 3$) For each factorization of $p-1$, that is, for each leaf of the tree in Figure 31, by Theorem 9, we need to test whether or not for all $p_j \in$ *pfactors*$(p-1)$, we have that $x^{(p-1)/p_j} \neq 1 \pmod p$.

For Point ($\beta 3$) we have the following three contributions to the computational cost.

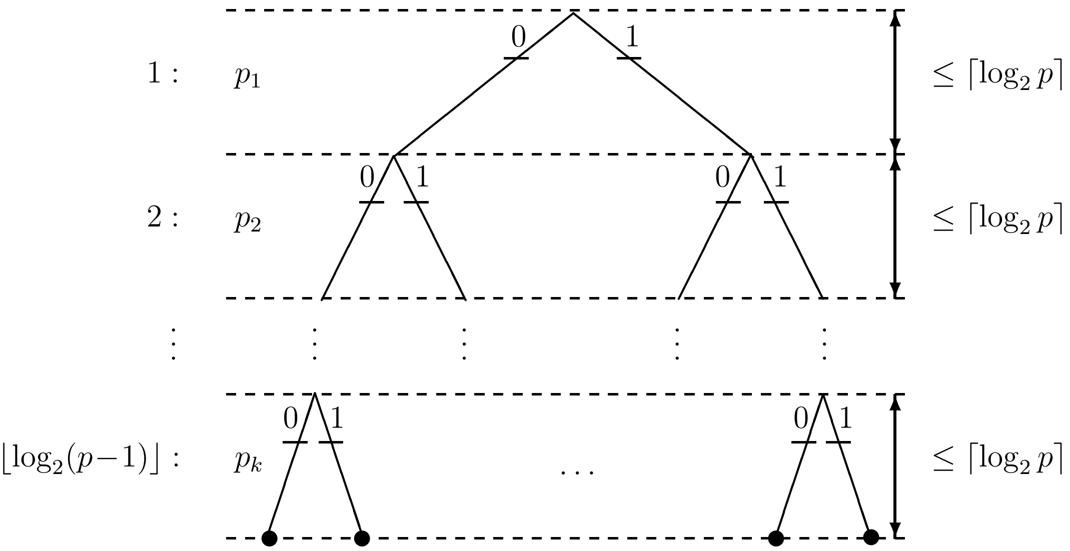

1 :
p_1
0
1
$\leq \lceil \log_2 p \rceil$
2 :
p_2
0
1
0
1
$\leq \lceil \log_2 p \rceil$
$\lfloor \log_2 (p-1) \rfloor$:
p_k
0
1
...
0
1
$\leq \lceil \log_2 p \rceil$

Fig. 31. Construction of the factorization of $p-1$. The tree is polynomially deep with respect to $\lceil \log_2 p \rceil$. Its depth is at most $\lfloor \log_2 (p-1) \rfloor \times \lceil \log_2 p \rceil$.

(β3.1) We compute, by successive squaring modulo p, the binary expansion of the powers modulo p of x, that is, the set $S = \{x,\ x^2 \pmod p,\ x^4 \pmod p, \ldots, x^{2^n} \pmod p\}$, where 2^n is the largest natural number such that $2^n \leq p-1$. We need to perform at most $\log_2 (p-1)$ multiplications modulo p.

(β3.2) We compute the values of $(p-1)/p_j$ for each $p_j \in \mathit{pfactors}(p-1)$. Since the cardinality of $\mathit{pfactors}(p-1)$ is at most $\log_2 p$, we need to perform at most $\log_2 p$ divisions modulo p.

(β3.3) For each value in the set $\{(p-1)/p_j \mid p_j \in \mathit{pfactors}(p-1)\}$ we add modulo p at most $\log_2 p$ powers of x occurring in S.

Thus, the total cost C_{tot} of the primality test is:

$$\begin{aligned} C_{tot} = {} & c_0 (\log_2 p)^3 && \text{(for } \alpha\text{)} \\ & + \Sigma_{1 \leq j \leq k}\, c\, (\log_2 p_j)^4 && \text{(for } \beta 1\text{)} \\ & + c_1 (\log_2 p)^3 && \text{(for } \beta 2 \text{ and } \beta 3.1\text{)} \\ & + c_2 (\log_2 p)^3 && \text{(for } \beta 3.2\text{)} \\ & + c_3 (\log_2 p)^3 && \text{(for } \beta 3.3\text{),} \end{aligned}$$

for some suitable positive constants c, c_0, c_1, c_2, and c_3.

Now, in order to conclude that the PRIME problem is in NP it is enough to observe that $C_{tot} < c\, (\log_2 p)^4$ for some suitable constant $c > 0$. This observation follows from the fact that:

- if $p-1 = p_1 \times \ldots \times p_k$ then $\log_2 (p-1) = \log_2 p_1 + \ldots + \log_2 p_k$, and
- for $h > 1$ and $p-1 > 2$ we have that:
  $(\log_2 p)^h > (\log_2 (p-1))^h > (\log_2 p_1)^h + \ldots + (\log_2 p_k)^h$. □

Note that it is well-known open problem to determine whether or not any problem in NP can be computed by a *deterministic* Turing Machine within polynomial time with respect to the size of the input. This problem is known as the 'P = NP problem'.

We have already seen that the minimal spanning tree problem can be solved in deterministic polynomial time by Kruskal's algorithm, but every algorithm which is currently known for solving the Hamiltonial circuit problem takes exponential time in a deterministic Turing Machine.

There are problems in NP which are *complete* (see Section 28 on page 128) in the sense that if a polynomial time, deterministic Turing Machine can be found for the solution of any NP-complete problem, then all problems in NP can be solved in polynomial time by a deterministic Turing Machine, and thus, P = NP.

In view of the fact that deterministic Turing Machines and IA-RAM's are polynomially related (see Section 31 on page 147), in the above sentence we can replace 'deterministic Turing Machine' by the informal notion of 'algorithm'.

In order to define the class of NP-complete problems in the following section we introduce the notion of problem reducibility.

27 Problem Reducibility

In Section 19 we have explained that via a suitable encoding, a problem A can be viewed as a language L_A on an alphabet Σ_A, that is, a problem A can be viewed as a subset of Σ_A^*. The language L_A can also be viewed as a predicate R_A on Σ_A^*, that is, a function from Σ_A^* to $\{$*false*, *true*$\}$ such that $w \in L_A$ iff $R_A =$ *true*.

Definition 5. [Problem Reduction in Polynomial-Time or Logarithmic-Space] Given a problem A associated with the language $L_A \subseteq \Sigma_A^*$ and a problem B associated with the language $L_B \subseteq \Sigma_B^*$, we say that problem A is *polynomial-time reducible* to problem B (or equivalently, predicate R_A is *polynomial-time reducible* to predicate R_B) and we write $A \rightarrow_p B$ (or $A \leq_p B$), iff there exists a deterministic Turing Machine M which for each w in Σ_A^*, produces, *in a polynomial time* with respect to the size of w, a word, call it $f(w)$, in Σ_B^*, such that $w \in L_A$ iff $f(w) \in L_B$.

Instead of polynomial-time reductions, we may introduce the *logarithmic-space reductions*, denoted $\leq_{logsp}$, by replacing in the above Definition 5 'polynomial-time reducible' by 'logarithmic-space reducible' and also 'in a polynomial-time' by 'by using logarithmic-space'. We will see the relationship between these two reductions in Section 28.1 on page 134.

It is easy to show that:

- if $A \rightarrow_p B$ and B is in the class P then A is in the class P, and

- if $A \rightarrow_p B$ and B is in the class NP then A is in the class NP.

(It is enough to observe that given any word $w \in L_A$, the size of the word $f(w) \in L_B$ is a polynomial of the size of w.)

Given a problem A which is polynomial-time reducible to a problem B, the notation $A \leq_p B$ reminds us that, informally speaking, 'A is below B in the level of difficulty'.

Given two problems A and B, if $A \leq_p B$ and $B \leq_p A$, then we say that the two problems are *polynomial-time interreducible*, and we write $A =_p B$.

The composition of two polynomial-time reductions is a polynomial-time reduction.

Two problems which are polynomial-time interreducible, are the following ones.

C-SATISFIABILITY

Input: a boolean formula F in conjunctive normal form (CNF), that is, a formula made out of *and*'s ($\wedge$), *or*'s ($\vee$), *not*'s ($\neg$), variables, and parentheses.

Output: 'yes' iff F is true for some truth assignment of the variables in F.

K-CLIQUE

Input: an undirected graph G with $n \geq 1$ nodes and an integer $k \geq 1$.

Output: 'yes' iff G has a k-clique, that is, a set of k nodes such that all $k\,(k-1)/2$ pairs of nodes are connected by an arc.

We will *not* formally prove the interreducibility between the C-SATISFIABILITY problem and the K-CLIQUE problem. That proof is left to the reader as an exercise. We will simply present the polynomial-time reduction of the C-SATISFIABILITY problem to the K-CLIQUE problem and the polynomial-time reduction of the K-CLIQUE problem to the C-SATISFIABILITY problem.

Let us start by constructing a function, call it g, such that the CNF formula F with k conjuncts is satisfiable iff $g(F)$ has a k-clique.

Let F be the conjunction $F_1 \wedge F_2 \wedge \ldots \wedge F_k$, where for $i = 1, \ldots, k$, the disjunction F_i is of the form: $x_{i,1} \vee \ldots \vee x_{i,r_i}$. The function g is defined as follows. $g(F)$ is the graph whose nodes are:

$\{[i, j] \mid 1 \leq i \leq k \text{ and } 1 \leq j \leq r_i\}$,

that is, each node corresponds to the occurrence $x_{i,j}$ of a literal (variables or negated variables) in F, and whose set of arcs is:

$\{\langle [i, j], [s, t] \rangle \mid i \neq s \text{ and } (x_{i,j} \neq \neg x_{s,t} \text{ or } \neg x_{i,j} \neq x_{s,t})\}$,

that is, an arc connects nodes x and y iff x and y belong to different conjuncts and they correspond to literals which are not negation of each other.

An example will clarify the ideas. Let us consider the formula $F = (a \vee \neg b) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)$. Let us consider the following three conjuncts of the formula F:

$F_1 = (a \vee \neg b)$, $F_2 = (b \vee \neg c)$, and $F_3 = (\neg a \vee \neg b \vee \neg c)$.

From F we can construct the graph $g(F)$ (see Figure 32). One can show that our given F is satisfiable iff $g(F)$ has a 3-clique.

It is not difficult to prove that for any F, the construction of the graph $g(F)$ can be done in polynomial time with respect to a suitable notion of the size of F, that is, for instance, with respect to the number of occurrences of literals in F.

Now we show how to reduce in polynomial time the K-CLIQUE problem to the C-SATISFIABILITY problem. Let us construct a function, call it f, such that the graph G has a k-clique iff $f(G)$ is satisfiable.

Let us assume that the n nodes of G are called $1, 2, \ldots, n$. We have that G has a k-clique iff there exists a total injection function h from $\{1, \ldots, k\}$ into $\{1, \ldots, n\}$. The function f translates this statement into a CNF boolean formula using the following $n^2 + n\,k$ variables:

$\{A_{ij} \mid 1 \leq i, j \leq n\} \cup \{H_{ij} \mid 1 \leq i \leq k \wedge 1 \leq j \leq n\}$

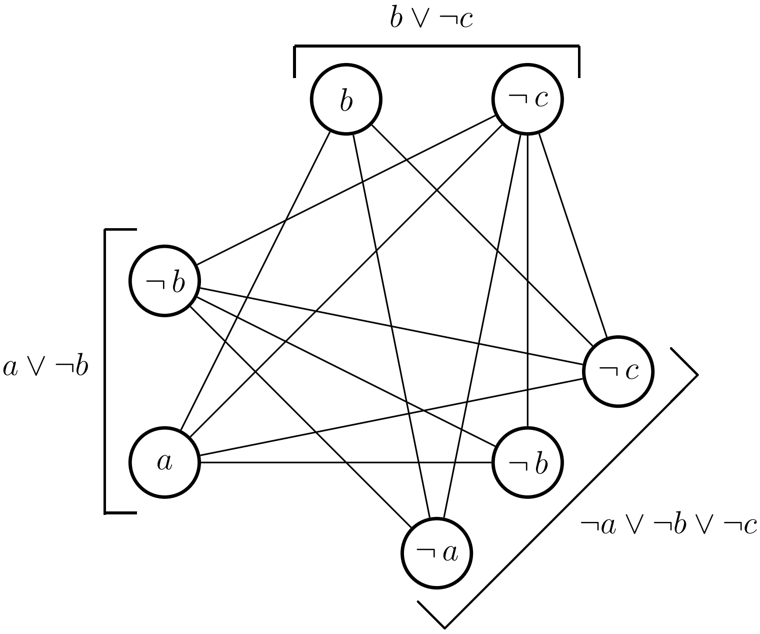


Fig. 32. The graph $g(F)$ corresponding to $F = (a \vee \neg b) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)$.

where A_{ij} is true iff $\langle i, j\rangle$ is an arc of G, and H_{ij} is true iff $h(i) = j$.
$f(G)$ is the formula $F = F1 \wedge F2 \wedge F3 \wedge F4 \wedge F5$, where:

$F1 = \bigwedge_{(i,j)\in G} A_{ij} \;\wedge\; \bigwedge_{(i,j)\notin G} \neg A_{ij}$

which ensures that the set $\{A_{ij} \mid 1 \leq i, j \leq n\}$ denotes the arcs of G

$F2 = \bigwedge_{1\leq i\leq k} (H_{i1} \vee \ldots \vee H_{in})$

which ensures that h is defined for all $i = 1, \ldots, k$,

$F3 = \bigwedge_{1\leq i\leq k,\, 1\leq j\neq m\leq n} \neg\, (H_{ij} \wedge H_{im})$

which ensures that h is single valued

$F4 = \bigwedge_{1\leq i\neq j\leq k,\, 1\leq m\leq n} \neg\, (H_{im} \wedge H_{jm})$

which ensures that h is an injection

$F5 = \bigwedge_{1\leq i\neq j\leq k,\, 1\leq s,t\leq n} ((H_{is} \wedge H_{jt}) \rightarrow A_{st})$

which ensures that for all i, j with $1 \leq i \neq j \leq k$, $\langle h(i), h(j)\rangle$ is an arc of G.

To put F in conjunctive normal form we need to recall that $(a \wedge b) \rightarrow c = \neg a \,\vee\, \neg b \,\vee\, c$ and $\neg(a \wedge b) = \neg a \,\vee\, \neg b$.

It is easy to prove that the transformation f can be done in polynomial time. For this proof one should replace the $\bigwedge$ conjunctions by the corresponding sequences of $\wedge$ conjunctions.

In order to show that the K-CLIQUE problem can be reduced in polynomial-time to the C-SATISFIABILITY problem one need to show that G has a k-clique for some $k \geq 1$ iff $f(G)$ is satisfiable.

Exercise 3. Verify that the C-SATISFIABILITY problem and the K-CLIQUE problem are both in NP. □

The reader should note that the definition of the reducibility between two problems, say A and B, makes reference to the pair of alphabets Σ_A and Σ_B, and to the representations of the input words. Strictly speaking, in order to state a reducibility relationship, one should also specify those alphabets and representations.

It is the case, however, that various standard representations are polynomial-time related, so that it is not necessary to specify under which representations the given problems are polynomial-time reducible. Care should be taken for the representation of the integers, where the integer n is usually assumed to be represented by $\log_2 n$ bits.

28 NP-Complete Problems

Definition 6. [NP-**complete Problems**] A given problem A is *NP-complete* iff A is in NP and for every other problem B in NP we have that $B \leq_p A$ (Cook, 1971).

Thus, we may say that NP-complete problems are the 'most difficult' problems in the class NP. Many well-known problems are NP-complete problems and thus, any two of them are polynomial-time interreducible.

If an NP-complete problem were shown to be in P then P = NP, because, by definition, every problem in NP is polynomial-time reducible to any NP-complete problem.

Let us list the following NP-complete problems [16].

1. C-SATISFIABILITY
Input: a CNF boolean formula F, that is, a boolean formula in conjunctive normal form.
Output: 'yes' iff F is satisfiable.

The size of the input formula is the number of the symbols of the formula.
The variant of the C-SATISFIABILITY problem, called SATISFIABILITY, where one does *not* insist that the given boolean formula be in conjunctive normal form, is also NP-complete.

2. 3-SATISFIABILITY
Input: a 3-CNF boolean formula F, that is, a boolean formula in conjunctive normal form with at most three literals in each disjunct. (By literal we mean either a variable or a negated variable.)
Output: 'yes' iff F is satisfiable.

3. K-CLIQUE
Input: an undirected graph G, with $n \geq 1$ nodes and an integer $k \geq 1$.
Output: 'yes' iff in G there exists in a k-clique, that is, a subgraph with k nodes such that there is an arc between any two nodes of that subgraph.

4. HAMILTONIAN CIRCUIT in a directed graph
Input: a directed graph G.
Output: 'yes' iff in G there is an Hamiltonian circuit, that is, a cycle which touches every node exactly once. Obviously, the first and the last node of the graph are the same node.

5. HAMILTONIAN CIRCUIT in a undirected graph
This problem is analogous to the problem of HAMILTONIAN CIRCUIT in a directed graph, but the input graph is now assumed to be an undirected graph and the circuit should be made out of undirected arcs.

6. VERTEX COVER
Input: an undirected graph G and an integer k.
Output: 'yes' iff there exists a set S of k nodes such that every arc of G has at least one of its two nodes in S.

7. COLOURABILITY
Input: an undirected graph G and an integer k.
Output: 'yes' iff each of the nodes can be assigned one of k colours such that no arc of G has its two nodes of the same colour.

8. SUBSET SUM
Input: a set $\{i_0, i_1, \ldots, i_r\}$ of positive integers.
Output: 'yes' iff there exists a subset S of $\{i_1, \ldots, i_r\}$ such that the sum of the integers in S is i_0.

9. NOSTAR-REG-EXPRESSIONS
Input: a regular expression E over a finite alphabet Σ not involving $*$ (that is, involving $\cdot$ and $+$ only), and an integer $k \geq 0$.
Output: 'yes' iff $E \neq \Sigma^k$ (that is, E denotes a language which is not Σ^k).

Note that in E the empty string ε is *not* allowed, because ε is an abbreviation for $\emptyset^*$.

10. COOK'S PROBLEM
Input: the description of a nondeterministic Turing Machine M over the input alphabet Σ such that M may run for a polynomial number of steps only, and an input string $w \in \Sigma^*$. (The polynomial is with respect to the size of w.)
Output: 'yes' iff M accepts w.

Cook's problem is NP-complete by definition.

Theorem 10. [**Cook's Theorem**, 1971] Cook's problem is polynomial-time reducible to the C-SATISFIABILITY problem.

Proof. Omitted. See [10]. □

The proof that the above problems are all NP-complete, can be found in [2,10,16]. Those proofs are based on the reductions indicated in Figure 33 on page 130.

In order to show that a given problem A is NP-complete it is enough to show that a NP-complete problem is polynomial-time reducible to A. For instance, it is enough to show that C-SATISFIABILITY is polynomial-time reducible to A. Note that in practice some reductions between NP-complete problems are much easier than others, thus, the choice of the problem which is already known to be NP-complete is important.

Now we prove that the 3-SATISFIABILITY problem is NP-complete by reduction of the C-SATISFIABILITY problem to 3-SATISFIABILITY problem. Let the given formula F be expressed by conjunctions of disjunctions. For each disjunction D of the form $(x_1 \vee \ldots \vee x_k)$ for $k > 3$, we consider the following formula:

$$E = (x_1 \vee y_1)\,(\neg y_1 \vee x_2 \vee y_2)\,(\neg y_2 \vee x_3 \vee y_3)\ \ldots\ (\neg y_{k-2} \vee x_{k-1} \vee y_{k-1})\,(\neg y_{k-1} \vee x_k)$$

where $y_1, y_2, \ldots, y_{k-1}$ are new variables, and for simplicity we used juxtaposition to denote conjunction.

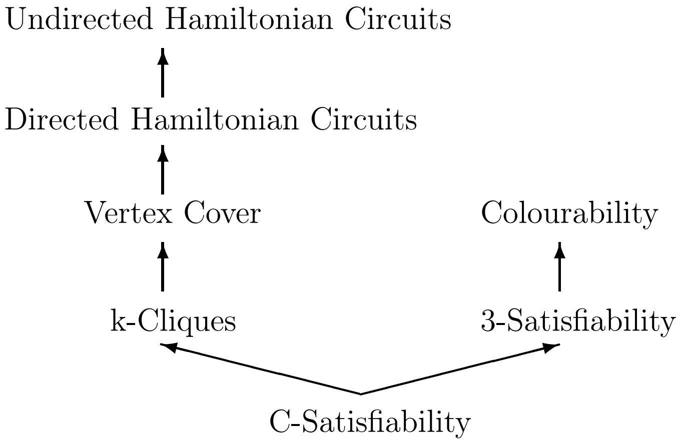


Fig. 33. Reducibility among NP-complete problems. The fact that $A \leq_p B$ is denoted by an arrow from A (drawn below) to B (drawn above).

We now show that for each assignment to the variables $x_1, \ldots, x_k$, there exists an assignment to the variables $y_1, y_2, \ldots, y_{k-1}$ such that E has value 1 (standing for *true*) iff D has value 1 (that is, one of the x_i's has value 1).

Indeed, let us assume that x_i has value 1. We set y_j to 1 for $j < i$, and y_j to 0 for $j \geq i$. Then E ha value 1. Conversely, if E has value 1 there are three cases: *either* (i) $y_1 = 0$, in which case x_1 has value 1, *or* (ii) $y_{k-1} = 1$, in which case x_k has value 1, *or* (iii) ($y_1 = 1$ and $y_{k-1} = 0$). In this last case we have that $E = (x_2 \vee y_2)\,(\neg y_2 \vee x_3 \vee y_3)\,\ldots\,(\neg y_{k-2} \vee x_{k-1}) = 1$, and this implies that at least one of x_i's has value 1. Indeed, if all x_i's have value 0 then for all possible assignments to $y_2, \ldots, y_{k-2}$, we have that:

$$(x_2 \vee y_2)\,(\neg y_2 \vee x_3 \vee y_3)\,\ldots\,(\neg y_{k-2} \vee x_{k-1}) = y_2\,(\neg y_2 \vee y_3)\,\ldots\,(\neg y_{k-3} \vee y_{k-2})\,(\neg y_{k-2}) = 0$$

The 2-SATISFIABILITY problem (where each conjunct is of at most two literals) is in P. This can easily be seen because by resolution of two conjuncts of at most two literals each, we get a new conjunct of at most two literals. Then the fact that the 2-SATISFIABILITY problem is in P follows from the fact that there are at most $O(n^2)$ different conjuncts of at most two literals each, being these literals taken from the set $\{x_1, \ldots, x_k\} \cup \{\neg x_1, \ldots, \neg x_k\}$.

Exercise 4. Show that the following so called Time-Tabling problem is NP-complete. Consider a set C of *candidates*, a set E of *examinations*, and for each candidate the set of examinations (subset of E) he/she has to take. To avoid clash, every candidate needs at least as many *examination periods* as the cardinality of the set of examinations he/she has to take. It is required to determine the minimum number of examination periods to avoid clashes for every candidate.

[*Hint*: equivalence to colourability.] □

Exercise 5. Consider the NOSTAR-REG-EXPRESSIONS problem over the alphabet $\{0, 1\}$.
Input: a regular expression E (with the operators $+$ and $\cdot$, and without the Kleene's star operator$*$) over $\{0, 1\}$ and an integer $k \geq 0$.
Output: 'yes' iff $E \neq \{0, 1\}^k$.

Consider the following function r from boolean expressions to regular expressions over $\{0,1\}$:
if $F = F_1 \wedge F_2 \wedge \ldots \wedge F_h$ with variables $\{x_1, x_2, \ldots, x_n\}$
then $r(F) = E_1 + E_2 + \ldots + E_h$, where for $i = 1, \ldots, h$, $E_i = e_{i1} \cdot e_{i2} \cdot \ldots \cdot e_{in}$, and for each $j = 1, \ldots, n$, we have that:
$e_{ij} = 0$ if x_j occurs in F_i
$e_{ij} = 1$ if $\neg x_j$ occurs in F_i
$e_{ij} = 1$ if neither x_j nor $\neg x_j$ occurs in F_i.
Show that C-SATIFIABILITY problem $\leq_p$ NOSTAR-REG-EXPRESSIONS problem. Deduce from this reduction that given two regular expressions E_1 and E_2 not involving $*$, the problem of answering 'yes' iff $E_1 \neq E_2$, is NP-complete. What happens when we allow $*$? (See the REG-EXPRESSIONS problem in Section 28.6 below. We have that the REG-EXPRESSIONS problem is PSPACE-complete with respect to polynomial-time reductions). □

Let us also consider the following NP-complete problems [30].

11. TRAVELING SALESMAN (TSP, for short).
Input: a weighted undirected graph G (weights are non-negative integers).
Output: 'yes' iff there exists a Hamiltonian circuit (that is, a circuit which touches every node exactly once) of minimal total weight.

A variant of this problem, called the k-Traveling Salesman problem (or k-TSP, for short), is defined as follows.
Input: a weighted undirected graph G and an integer $k \geq 0$.
Output: 'yes' iff there exits a Hamiltonian circuit in G whose weight is less or equal k.
Also this problem is NP-complete.

The variant of the TSP in which we insist that G be a complete graph (that is, there is an arc between any two nodes), is NP-complete as well. Indeed, TSP is an instance of this variant if some weights are 0.

12. INTEGER LINEAR PROGRAMMING (ILP, for short).
Input: an $n{\times}m$ matrix A of *integers*, and a column vector b $(n \times 1)$ of *integers* (the integers in A and b are not necessarily positive).
Output: 'yes' iff there exists a column vector x of $m \times 1$ *integers* such that $Ax \geq b$ and $x \geq 0$ (that is, each component of x is a non-negative integer). (An equivalent formulation of the ILP problem is the one where we impose the condition $Ax = b$, instead of $Ax \geq b$.)

A variant of the ILP problem is as follows. We have in input also a row c of $1 \times m$ integers (not necessarily positive) and we give the answer 'yes' iff the column vector x also minimizes the product $c\,x$. Also this variant is NP-complete.

Starting from the first definition of the ILP problem with either $Ax \geq b$ or $Ax = b$, if we do not insist that x be made out of integers, the problem is called LINEAR PROGRAMMING (LP, for short), and it is in P [21].

13. BINARY PARTITION
Input: a set S of positive integers $c_1, \ldots, c_r$.
Output: 'yes' iff there exists a subset A of S such that the sum of the elements in A is equal to the sum of the elements in $S - A$.

14. KNAPSACK
Input: a set S of positive integers $c_1, \ldots, c_r$ (called *sizes*), a set V of positive integers $v_1, \ldots, v_r$ (called *values*), and a positive integer B (called *bin capacity*).
Output: 'yes' iff there exists a subset $S1$ of S and a corresponding (that is, with the same subscripts) subset $V1$ of V such that the sum of the sizes is not larger than B and the sum of the values is maximal (among all possible choices of the subset $S1$ and the corresponding subset $V1$).

A variant of the Knapsack problem, called the k-KNAPSACK problem, is as follows.
Input: a set S of positive integers $c_1, \ldots, c_r$ (called *sizes*), a set V of positive integers $v_1, \ldots, v_r$, (called *values*), a positive integer B (called *bin capacity*), and a positive integer k.
Output: 'yes' iff there exists a subset $S1$ of S and a corresponding (that is, with the same subscripts) subset $V1$ of V such that the sum of the sizes is not larger than B and the sum of the values is not smaller than k.

Also this problem is an NP-complete problem.

15. BIN PACKING
Input: a set S of positive integers $c_1, \ldots, c_r$, and a positive integer bin capacity B.
Output: 'yes' iff there exists a partition of S in the minimum number of disjoint subsets such that the sum of the integers in each subset is not greater than B.

A variant of the BIN PACKING problem, called the k-BIN PACKING problem, is as follows.
Input: a set S of positive integers $c_1, \ldots, c_r$, a positive integer bin capacity B, and a positive integer k.
Output: 'yes' iff there exists a partition of S in k disjoint subsets such that the sum of the integers in each subset is not greater than B.

Also this problem is NP-complete.

Note 1. In the Problems 11–15 above, instead of giving the answer 'yes', we may as well give the witness of that answer. For instance, in the case of the TRAVELING SALESMAN PROBLEM we may give the Hamiltonian circuit with minimal weight. □

One can argue that the k-Traveling Salesman problem, the k-KNAPSACK problem, and k-BIN PACKING problem are in NP from the fact that the so called corresponding 'optimization variants' where one has to find minimal or maximal solutions, are in NP by reducing the optimization variants to a polynomial number of the k-variants as we now indicate. We assume that we can compute in polynomial time (w.r.t. the size of the input) a polynomial bound (w.r.t. the size of the input), call it d_0, for the value of k.

We solve the optimization variant by solving the k-variants for $k = d_0$, $k = d_1$, $k = d_2$, and so on, where the values of d_1, d_2, etc., are obtained by binary search, starting from d_0, until the optimal solution is found. By doing this, we need to solve at most $O(\log_2 d_0)$ k-variants for finding the optimal solution, and thus, we remain within polynomial-time cost (w.r.t. the size of the input).

The reader may notice that the k-variants can be associated with a language in a more direct way than the corresponding optimization variants. For instance, in the case of the

k-TSP it is easy to consider a word for each encoding of a Hamiltonian circuit whose total weight is less or equal to k.

In order to show that a problem in NP-complete one may also use the so called *restriction technique*.

For instance, NP-completeness of the k-Traveling Salesman problem is derived from the fact that the set of all instances of the k-Traveling Salesman problem such that all weights are 1's, for k sufficiently large constitutes the Hamiltonian circuits problem in undirected graphs.

NP-completeness of the k-KNAPSACK problem is derived from the fact that the set of all instances of the k-KNAPSACK problem such that for $i = 1, \ldots, r$, $c_i = v_i$ and $B = k = (\Sigma_i\, c_i)/2$ constitutes the BINARY PARTITION problem.

Analogously, the BIN PACKING problem can be shown to be NP-complete, because the set of all its instances such that $B = (\Sigma_i\, c_i)/2$ constitutes the BINARY PARTITION problem. Indeed, (i) to check whether or not there is a solution for $k = 2$, corresponds to solve the BINARY PARTITION problem, and (ii) to minimize k, that is, the number of disjoint subsets, it is required to get as close as possible to $k = 2$, and since $k = 1$ is impossible (because $B = (\Sigma_i\, c_i)/2$), one needs to check whether or not there is a solution for $k = 2$.

Recall that, even if the answer to *some* instances of the problems we have mentioned, can be determined within polynomial time (like, for instance, for BINARY PARTITION when the sum of the weights is odd, or in the case of BIN PACKING, when $B < max\, \{c_i\}$), this does not necessarily make the problem not to be NP-complete, because for a problem to be in P we require that its solution should be determined within polynomial time for *every* input.

Exercise 6. Show that BINARY PARTITION is in NP.
[*Hint.* Let S be $\{c_1, \ldots, c_r\}$. Consider a full binary tree where for $h = 1, \ldots, r$, at level h it is decided whether or not c_h should be in the set A to be found. In every node at level r we determine whether or not the sum of the chosen integers is equal to the sum of the ones in S which are not chosen.] □

Exercise 7. Show that the language of the boolean expressions (with $\wedge, \vee, \neg$, and variables) which are *not* tautologies is NP-complete [2, page 400]. □

Exercise 8. Show that the problem of determining whether a regular expression over $\Sigma = \{0\}$ does not denote 0^* is NP-complete [2, page 401]. □

The reader should notice that the notion of NP-completeness does not imply that problems are either polynomial-time or exponential-time in the sense that they may be in between, that is, they could require $O(n^{\log_2 n})$ time.

Note 2. Recall that for any constant $a > 0$ and $b > 1$ we have: $O(n^a) \subset O(n^{\log_2 n}) \subset O(b^n)$, in the sense that:

$n^a = O(n^{\log_2 n})$ for $a > 0$, and $n^{\log_2 n} = O(b^n)$ for any $b > 1$, while
$n^{\log_2 n} \neq O(n^a)$ for $a > 0$, and $b^n \neq O(n^{\log_2 n})$ for any $b > 1$. □

If every problem in NP is polynomial-time reducible to a problem in DTIME($T(n)$) then NP $\subseteq \bigcup_{i>0}$ DTIME($T(n^i)$), because a polynomial-time reduction from problem A to problem B cannot generate an input for problem B which is more than a polynomial of the size of the input of problem A.

This statement holds also if we replace 'polynomial-time reducible' by 'logarithmic-space reducible' (see Definition 5 on page 125). In the following Section 28.1 we will say more about the relationship between these two notions of reducibility.

28.1 Polynomial-Time versus Logarithmic-Space Reductions

In Definition 5 of Section 27 we have introduced two concepts of reducibility between problems: (i) the polynomial-time reducibility, denoted $\leq_p$, and (ii) the logarithmic-space reducibility, denoted $\leq_{logsp}$. Recall that in order to define the notion of logarithmic-space reducibility we consider, instead of a deterministic Turing Machine, a deterministic, off-line Turing Machine M (see Section 4.4 on page 27 and Section 25 on page 113). We assume here that such an off-line deterministic Turing Machine has a read-only *input tape* (the head moves to the right and to the left), together with a write-only *output tape* (where the head moves to the right only and always reads the blank symbol), and a *working tape* (where, as usual, the head may move to the left and to the right, and it may read and write). As indicated in Theorem 2 of Section 25 on page 114, we may equivalently have $k\,(\geq 1)$ working tapes, instead of one only. Moreover, with respect to Section 25 we assume here, as some authors do, that the off-line deterministic Turing Machine M has an output tape. Note, however, that this assumption does not modify the computational power of the machine M because the output tape can be seen as an extra working tape (with restrictions on the allowed operations).

Theorem 11. *If a problem A is logarithmic-space reducible to a problem B (that is, $A \leq_{logsp} B$), then problem A is also polynomial-time reducible to problem B (that is, $A \leq_p B$).*

Proof. It is enough to show the following implication (α): *if* an off-line deterministic Turing Machine M when solving the problem A with input word w, can use in the working tape only a number of cells which is proportional to $\log_2(size(w))$, *then* M can determine at most a polynomial number of different 5-tuples whose components are: (i) the internal state of M, (ii)–(iv) the positions of the heads in the input tape, the working tape, and the output tape, and (v) the content of the whole working tape. Having proved the implication (α), and taking into account that the moves of M are determined by the values of these 5-tuples, we will have that: *either* M stops within a polynomial number of steps *or* it loops. Since we have assumed (see Definition 5 on page 125) that M given an instance i_A of the problem A, generates an instance i_B of the problem B (so that the answer to i_B is also the answer to i_A), then M will do so in polynomial-time.

Now, we will prove the above implication (α). Let Q denote the set of the internal states and Γ denote the set of the tape symbols. We have that the number of different 5-tuples is bounded by the following product P for some values of the constants c and d:

$P = |\,Q\,|$ (for the internal state)
$\times$ *size*(w) (for the position of the head in the input tape)
$\times\ c\ \log_2(\mathit{size}(w))$ (for the position of the head in the working tape)
$\times$ 1 (for the position of the head in the output tape)
$\times\ |\,\Gamma\,|^{\,d\ \log_2(\mathit{size}(w))}$ (for the content of the working tape).

Note that: (i) the input tape cannot be written and thus, it contributes to the product P by a factor *size*(w) (and not $|\,\Gamma\,|^{\mathit{size}(w)}$), and (ii) the head on the output tape always scans a cell with a blank symbol and thus, it contributes to the product P by a factor 1 only. We have that P is a polynomial of *size*(w), because:

$$a^{d\ \log_b s} = s^{\,d\ \log_b a} \quad \text{for } a, d, s > 0 \text{ and } b \neq 1.$$

This completes the proof. □

As a consequence of this theorem, we have that:
(i) if $A \leq_{\mathit{logsp}} B$ and B is in the class P then A is in the class P, and
(ii) if $A \leq_{\mathit{logsp}} B$ and B is in the class NP then A is in the class NP.

We also have that the compositions of two logarithmic-space reductions is a logarithmic-space reduction.

Note that if we use the logarithmic-space reductions, instead of polynomial-time reductions, we get a different notion of NP-completeness.

However, as shown in [16], the problems listed above, that is, Satisfiability, C-Satisfiability, 3-Satisfiability, k-Clique, Hamiltonian Circuit (in directed and undirected graphs), Vertex Cover, Colourability, Subset Sum, Nostar-Reg-Expressions, Cook's problem, Traveling Salesman problem, Binary Partition, Bin Packing, Knapsack, Integer Linear Programming are all NP-complete *also* with respect to logarithmic-space reductions.

If a problem A is NP-complete with respect to logarithmic-space reductions and we have that A is in DSPACE(log n), then NP = DSPACE(log n).

Note that if all problems in NP are polynomial-time reducible to a problem A and we have that A is in DSPACE(log n) then we cannot conclude that NP = DSPACE(log n) (because polynomial-time reduction from a problem B_1 to a problem B_2 does *not* imply a logarithmic-space reduction from problem B_1 to problem B_2).

28.2 NP-Complete Problems and Strongly NP-Complete Problems

Let us consider the following three problems: BINARY PARTITION, KNAPSACK, and BIN PACKING. The first two differ from the last one, because they are *not strongly* NP-complete, while the last one is *strongly* NP-complete.

Definition 7. [Strong NP**-complete Problems]** A problem is said to be *strongly* NP-*complete* iff it is NP-complete also when the numbers involved are represented in unary (and *not* in binary) notation.

Here is a Dynamic Programming technique for solving the BINARY PARTITION problem. This technique shows that BINARY PARTITION is *not* strongly NP-complete.

Let the given set S of positive integers be $\{c_1, \ldots, c_r\}$. We first compute the sum B of the elements in S. Then we construct the following sequence s of lists of integers: $A_0, A_1, \ldots, A_r$, where the list A_0 is made out of 0 only (that is, $A_0 = [0]$), and for any

$i > 0$, the list A_i is obtained by adding c_i to each element of the lists $A_0, A_1, \ldots, A_{i-1}$ and then by concatenating the resulting lists together. In particular, if $r \geq 3$ we have that: $A_0 = [0]$, $A_1 = [c_1]$, $A_2 = [c_2,\, c_1+c_2]$, and $A_3 = [c_3,\, c_1+c_3,\, c_2+c_3,\, c_1+c_2+c_3]$. While performing the concatenation of those lists we also erase the duplicated elements and the integers larger than $B/2$. We return 'yes' iff $B/2$ occurs in the sequence s.

In each list A_i, for $0 \leq i \leq r$, we have to consider up to $B/2 + 1$ different integers, because in $[A_0, A_1, \ldots, A_{i-1}]$ there are at most $B/2 + 1$ integers (The term '+1' is due to the presence of the integer 0), each integer being of at most $\log_2(B/2)$ bits, and this is done at most r times. Thus, in the worst case, the total number of steps to construct the sequence s is at most $r \times (B/2 + 1) \times \log_2(B/2)$. This is a polynomial of the size of the input when the input is written in unary. In this case, in fact, the size of the input is: $c_1 + \ldots + c_r$, which is equal to B, and we also have that $r \leq B$.

However, $r \times (B/2 + 1) \times \log_2(B/2)$ is *not* a polynomial of the size of the input, when the input is written in binary, in which case the size of the input is $\log_2 c_1 + \ldots + \log_2 c_r$ which is at most $r \times (\log_2 B)$. Indeed, it does not exist any polynomial of $r \times (\log_2 B)$ which is asymptotically above $r \times B$ (because it is not the case that there exist k, r_0, $B_0 \geq 0$ such that for all $r > r_0$ and for all $B > B_0$ we have that: $rB < (r \log_2 B)^k$).

28.3 NP Problems and co-NP Problems

It is an open problem to determine whether or not the class NP is closed under complementation, that is, whether or not NP = co-NP.

If NP ≠ co-NP, then P ≠ NP (because we have that P = co-P, that is, the class P of languages (or problems) is closed under complementation). The converse does *not* hold, that is, P ≠ NP does not imply that NP ≠ co-NP, and thus, it may be the case that P ≠ NP and NP = co-NP [13, page 182].

Theorem 12. NP is closed under complementation iff we have that the complement of some NP-complete problem is in NP.

Proof. (*only-if part*) It is obvious. (*if part*) Let us consider an NP-complete problem p and let us assume that its complement, call it co-p, is also in NP. Let us consider a problem b in NP. We have to show that also its complement co-b is in NP. First, we have that b is polynomial-time reducible to p because p is NP-complete. We also have that co-b is polynomial-time reducible (by the same reduction) to co-p, which is in NP. We conclude that co-b is in NP, as we wanted to show. Note that this proof is valid both for polynomial-time reductions and logarithmic-space reductions. □

28.4 Complete Problems and Hard Problems with respect to a Class of Problems

In Section 24 we have introduced the notions of hard and complete subsets of the set of natural numbers with respect to 1-reducibility between sets (see Definition 13 on page 110). Now we will introduce the very similar notions of hard and complete problems of a class of problems.

The notions of hard and complete subsets were based on the 1-reducibility relationship between sets (see again Definition 13 on page 110). Similarly, the notions we will introduce in the following definition are based on reducibility relationships between problems. In Definition 5 on page 125 we have already introduced two such relationships: the polynomial-time reductions and the logarithmic-space reductions.

Definition 8. [Hard and Complete Problems] Let us consider a class C of problems and a set R of reductions between problems. We say that a problem p is *C-hard* with respect to the set R iff every problem in C can be reduced via a reduction in R to the problem p.

We say that a problem p is *C-complete* with respect to the set R iff (i) every problem in C can be reduced via a reduction in R to the problem p, and (ii) $p \in C$.

Thus, a problem p is NP-complete with respect to the polynomial-time (or logarithmic-space) reductions iff it is NP-hard with respect to the polynomial-time (or logarithmic-space) reductions and it is in NP.

For instance, we have that ILP (Integer Linear Programming) is NP-complete with respect to logarithmic-space reductions, and this can be shown by proving that: (i) ILP is NP-hard with respect to logarithmic-space reductions, and (ii) ILP is in NP [16, page 338].

28.5 Approximation Algorithms

There are algorithms which take polynomial-time and solve in an approximate way NP-complete problems in the sense we will now explain [16,30]. Consider, for instance, the METRIC TRAVELING SALESMAN problem (denoted ΔTSP), that is, the TRAVELING SALESMAN problem with the hypothesis that the given distances satisfy the triangle inequality: $d(a,c) \leq d(a,b) + d(b,c)$.

ΔTSP is NP-complete when the input is a weighted undirected graph G and an integer $k \geq 0$, and the output is a Hamiltonian circuit whose weight is less or equal k.

For ΔTSP there exists an algorithm due to Christofides, which finds a solution, which is within 3/2 of the cost of the optimal solution [30, page 416].

Note, however, that if the hypothesis on the triangle inequality is *not* assumed (that is, if we consider the standard TRAVELING SALESMAN problem) then the existence of a polynomial-time algorithm which finds a circuit, which is within twice the cost of the optimal circuit, implies that $\mathrm{P} = \mathrm{NP}$.

28.6 Stratification of Complexity Classes

Figure 34 on page 139 gives a pictorial view of the inclusions of some classes of problems (denoted in bold, capital letters) of different computational complexity. If a class **A** is below class **B** then $\mathbf{A} \subseteq \mathbf{B}$. With reference to that figure we recall the following points.

As indicated in Section 25 on page 115, it is an open problem
(i) whether or not P = NP (see end of Section 26 on page 26),
(ii) whether or not NP = PSPACE [19, page 98],
(iii) whether or not PSPACE = EXPTIME [19, page 102], and
(iv) whether or not EXPTIME = NEXPTIME [19, page 103].

Let EXPTIME be $\bigcup_{k>0}\mathrm{DTIME}(2^{n^k})$, that is, the set of all problems solvable in time $O(2^{p(n)})$, where $p(n)$ is a polynomial in the size n of the input. Let 2-EXPTIME be

$\bigcup_{k>0}$DTIME($2^{2^{n^k}}$), that is, the set of all problems solvable in time $O(2^{2^{p(n)}})$, where $p(n)$ is a polynomial in the size n of the input. In general, for any $k \geq 2$, k-EXPTIME is the set of all problems solvable in time $O(2^{\cdot^{\cdot^{2^{p(n)}}}})$, where $p(n)$ is a polynomial in the size n of the input and there are k occurrences of 2 in the ladder of 2's.

The class ELEMENTARY $=_{def} \bigcup_{k>1}$ k-EXPTIME is a *proper subset* of REC.

We have the following inclusions (see also Section 29.1 on page 142 and Section 29.2 on page 143):

PSPACE $\subseteq$ EXPTIME $\subseteq$ EXPSPACE $\subseteq$ 2-EXPTIME $\subseteq$ 2-EXPSPACE $\subseteq$
$\subseteq$ 3-EXPTIME $\subseteq$ and so on.

It is also an open problem whether or not P = PSPACE. If P = NP then EXPTIME = NEXPTIME [19, page 103].

We have that: P $\subseteq$ NP $\subseteq$ PSPACE (= NSPACE) $\subseteq$ EXPTIME

where at least one of these containments is proper, because we have that P $\subset$ EXPTIME as a consequence of the Deterministic Time Hierarchy Theorem 14 on page 142.

We have that: DSPACE(log n) $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE (= NSPACE), where at least one of these containments is proper, because we have that DSPACE(log n) $\subseteq$ NSPACE(log n) $\subset$ PSPACE (= NSPACE) as a consequence of the Nondeterministic Space Hierarchy Theorem 17 on page 144. We also have that: NP $\subset$ NEXPTIME (see Theorem 18 on page 144). It is an open problem whether or not PSPACE = NEXPTIME (see also what we have stated on page 161 at end of Section 33).

It may be that P = NP and NP $\subset$ PSPACE (proper containment).

If P $\neq$ NP then it is undecidable to determine whether or not a given language L is in P, when it is known that L is in NP.

If there exists an NP-complete language $L \subseteq 0^*$ then P = NP [16, page 369].

The class REC (short for recursive) is the class of the decidable problems. Sometime, in the literature that class is also called DECIDABLE. This terminology is consistent with the one we have used in Definition 13 on page 78 at the end of Section 14.

Now we give the formal definitions of the theories and problems which are indicated in Figure 34 on page 139.

(A) The WS1S theory is the *Weak Monadic Second Order Theory of Successor* (1 in WS1S stands for 1 successor). It is a theory of the natural numbers with 0 (zero), s (successor), $<$, $=$, the usual propositional connectives, comma, parentheses, existential and universal quantifiers, individual variables (ranging over natural numbers), and a countably infinite set of monadic predicate variables, called *set variables* (ranging over finite sets of natural numbers).

If F is a set variable, we write $x \in F$, instead of $F(x)$. For instance, the following is a sentence of WS1S:

$$\exists F\, \exists x\, \forall y\, (x \in F \;\wedge\; (y > x \rightarrow y \notin F)).$$

This sentence is true and it says that there exists a non empty subset with a largest element.

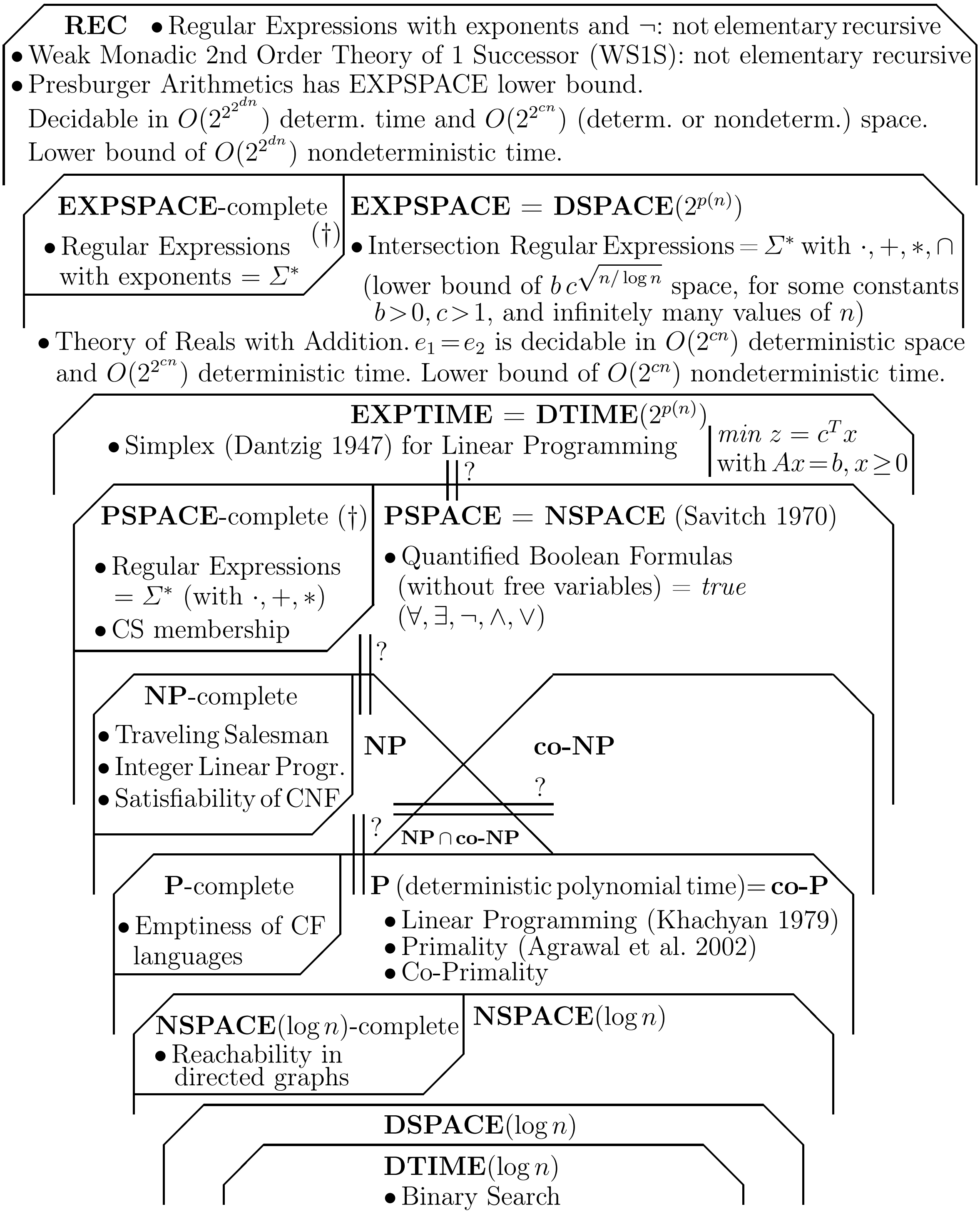

REC • Regular Expressions with exponents and ¬: not elementary recursive
• Weak Monadic 2nd Order Theory of 1 Successor (WS1S): not elementary recursive
• Presburger Arithmetics has EXPSPACE lower bound.
Decidable in $O(2^{2^{2^{dn}}})$ determ. time and $O(2^{2^{cn}})$ (determ. or nondeterm.) space.
Lower bound of $O(2^{2^{dn}})$ nondeterministic time.
EXPSPACE-complete (†)
• Regular Expressions with exponents = Σ^*
EXPSPACE = **DSPACE**$(2^{p(n)})$
• Intersection Regular Expressions = Σ^* with $\cdot, +, *, \cap$
(lower bound of $b\,c^{\sqrt{n/\log n}}$ space, for some constants $b>0, c>1$, and infinitely many values of n)
• Theory of Reals with Addition. $e_1 = e_2$ is decidable in $O(2^{cn})$ deterministic space and $O(2^{2^{cn}})$ deterministic time. Lower bound of $O(2^{cn})$ nondeterministic time.
EXPTIME = **DTIME**$(2^{p(n)})$
• Simplex (Dantzig 1947) for Linear Programming
min $z = c^T x$
with $Ax = b, x \geq 0$
|| ?
PSPACE-complete (†)
• Regular Expressions = Σ^* (with $\cdot, +, *$)
• CS membership
PSPACE = **NSPACE** (Savitch 1970)
• Quantified Boolean Formulas (without free variables) = *true* $(\forall, \exists, \neg, \wedge, \vee)$
|| ?
NP-complete
• Traveling Salesman
• Integer Linear Progr.
• Satisfiability of CNF
NP
co-NP
?
|| ?
NP ∩ **co-NP**
P-complete
• Emptiness of CF languages
P (deterministic polynomial time)= **co-P**
• Linear Programming (Khachyan 1979)
• Primality (Agrawal et al. 2002)
• Co-Primality
NSPACE$(\log n)$-complete
• Reachability in directed graphs
NSPACE$(\log n)$
DSPACE$(\log n)$
DTIME$(\log n)$
• Binary Search

Fig. 34. Inclusions of some computational complexity classes. $p(n)$ is a polynomial in n. The complete classes marked with (†) are complete w.r.t. polynomial-time reductions. The other complete classes are complete w.r.t. logarithmic-space reductions (and, by Theorem 11 on page 134, also w.r.t. polynomial-time reductions). We drew neither the class **NEXPTIME** nor the class **ELEMENTARY**. We have that:
EXPTIME $\subseteq$ **NEXPTIME** $\subseteq$ **EXPSPACE** $\subset$ **ELEMENTARY** $\subset$ **REC**.
The symbols $\overset{?}{=\!=}$ and $||\,?$ relate two classes whose equality is an open problem.

Given any algorithm for deciding WS1S (and there exist algorithms for doing so), for any $k \geq 0$ there exists a formula of length n for which that algorithm takes time:

$$2 \uparrow (2 \uparrow \ldots (2 \uparrow n) \ldots)$$

where: (i) ↑ denotes exponentiation, and (ii) 2 occurs k times in the above expression. The same holds for the space requirements of formulas in WS1S [27]. Thus, it does not exist any m such that the decidability problem of the WS1S theory is in $\bigcup_{k<m} k$-EXPTIME. We state this fact by saying that the decidability problem of the WS1S theory is *not elementary recursive*.

(B) The *regular expressions with exponents and negation* over the alphabet Σ are the regular expressions with the usual operators $\cdot, +$, and $*$, where we also allow exponents (written in binary) and negation. For instance, the expression $(a^{11}+b)^*$ with the exponent 11 (which is 3 in binary) denotes the set $\{\varepsilon, aaa, b, aaaaaa, aaab, baaa, bb, \ldots\}$, and the language $L(\neg e)$ denoted by the regular expression $\neg e$ is $\Sigma^* - L(e)$.

The equivalence problem between two regular expressions with exponents and negation is decidable, but it is not *elementary recursive*, that is, it does not exist any m such that this equivalence problem is in $\bigcup_{k<m} k$-EXPTIME [29, page 504].

(C) *Presburger Arithmetics* is the theory of (positive, negative, and 0) integers with addition $(Z, +, =, <, 0, 1)$. It is decidable and its decision procedure requires at least nondeterministic time $O(2^{2^{c\,n}})$ for some $c>0$, infinitely often, that is, for infinitely many n, where n is the size of the formula. The relation symbol $<$ can be avoided if we consider only non-negative integers, because $s < t$ holds iff $\exists x\,(s+x = t)$. An example of a formula of Presburger Arithmetics is:

$$\forall x\, \exists y\, \exists z\, (x + z = y \;\wedge\; \exists w\, (w + w = y))$$

Presburger Arithmetics can be decided in deterministic time $O(2^{2^{2^{dn}}})$ for some $d>0$. It can be decided in (deterministic or nondeterministic) space $O(2^{2^{cn}})$ for some $c > 0$ [16, page 371].

Recall that, in contrast with Presburger Arithmetics, the set of all true statements in the *Theory of Natural Numbers* $(N, +, \times, =, <, 0, 1)$ (also called *Peano Arithmetics*) is undecidable (This result is stated in the famous Incompleteness Theorem proved by K. Gödel).

(D) Let us consider the following problem.

- INTERSECTION-REG-EXPRESSIONS

Input: an intersection regular expression (called *semi-extended regular expression* in [2, page 418]) E, that is, a regular expression E with the operators: $\cdot, +, *, \cap$ (but not $\neg$) over the alphabet Σ.
Output: 'yes' iff $E = \Sigma^*$.

To solve this problem it takes for any expression E of size n, at least $bc^{\sqrt{(n/\log n)}}$ space (and time), for some constants $b>0$, $c>1$, and for infinitely many values of n.

(E) The *Theory of Real Numbers* with addition $(R, +, =, <, 0, 1)$ is decidable and to decide whether or not $e_1 = e_2$ we need nondeterministic exponential time, that is, there exists a formula of size n, such that any nondeterministic algorithm takes at least 2^{cn} steps, for some constant $c>0$ and for an infinite number of n's. Thus, the decidability of the Theory of Real Numbers with addition is in the $\Omega(2^{cn})$ time complexity class.

The Theory of Real Numbers with Addition is decidable in doubly exponential time, that is, $O(2^{2^{cn}})$, and exponential space, that is, $O(2^{cn})$. This theory is also nondeterministic exponential time-hard with respect to polynomial-time reductions.

(F) There are instances of the simplex method (due to G. B. Danzig in 1947) [7] for solving LINEAR PROGRAMMING, for which the required time is exponential.

(G) Let us consider the set of regular expressions *with exponents* over the alphabet Σ, that is, expressions constructed from the symbols in Σ and the usual operators: $\cdot, +, *$, and also the operator $\uparrow n$ (with n written in binary), for $n \geq 0$. For instance, $(a+b)\uparrow 101$ over the alphabet $\{a, b\}$ denotes the expression (because $5 = 101_2$):

$(a+b) \cdot (a+b) \cdot (a+b) \cdot (a+b) \cdot (a+b)$.

In Figure 34 on page 34 we have also considered the following problems.

• REG-EXPRESSIONS WITH EXPONENTS
Input: a regular expression E with exponents over the alphabet Σ. The exponents are written in binary.
Output: 'yes' iff $E = \Sigma^*$.
This problem is EXPSPACE-complete with respect to polynomial-time reductions [16, page 353].

• QUANTIFIED BOOLEAN FORMULAS without free variables (QBF, for short).
Input: a quantified boolean formula F (propositional variables: $P1, P2, \ldots$, connectives: and, or, not, quantifiers $\forall, \exists$) without free variables.
Output: 'yes' iff $F = \mathit{true}$.
Since the SATISFIABILITY problem is a particular QBF problem, the QBF problem is NP-hard with respect to logarithmic-space reductions. We have that the QBF problem is PSPACE-complete with respect to polynomial-time reductions.

• CONTEXT-SENSITIVE MEMBERSHIP.
Input: a context-sensitive grammar G and a word $w \in \Sigma^*$.
Output: 'yes' iff $w \in \Sigma^*$.
This problem is in NSPACE(n), that is, (by Savitch's Theorem 16 on page 143) in DSPACE(n^2). It is also PSPACE-complete with respect to polynomial-time reductions.

• REG-EXPRESSIONS.
Input: a regular expression E over the alphabet Σ, with the operators: $\cdot, +$, and $*$ only.
Output: 'yes' iff $E = \Sigma^*$.
This problem is PSPACE-complete with respect to polynomial-time reductions.
Note that the NOSTAR-REG-EXPRESSIONS problem is NP-complete with respect to logarithmic-space reductions. We have already proved in Exercise 5 on page 130 above that it is NP-complete with respect to polynomial-time reductions.

• EMPTINESS FOR CONTEXT-FREE LANGUAGES.
Input: a context free grammar G.
Output: 'yes' iff $L(G) = \emptyset$.
This problem is P-complete with respect to logarithmic-space reductions.

• REACHABILITY IN DIRECTED GRAPHS.
Input: a directed graph G with nodes $1, \ldots, n$, and two nodes i and j.
Output: 'yes' iff there is a path from i to j.
This problem is NSPACE(log n)-complete with respect to logarithmic-space reductions.

29 Computational Complexity Theory

We briefly recall here some results concerning the Theory of Computational Complexity.

29.1 Deterministic Space and Deterministic Time Hierarchies

By enlarging the time and space limits the class of recognized languages increases, as the following theorems show.

Definition 9. [Space Constructable and Fully Space Constructable Functions] A function $S(n)$ is said to be *space constructible* iff there exists a Turing Machine M which for each input of size n, computes the result by using at most $S(n)$ cells, and for *some* input of size n, M actually uses $S(n)$ cells.

If for *all* input of size n, M actually uses $S(n)$ cells, we say that $S(n)$ is *fully space constructible*.

If a function $S(n)$ is space constructible and for all $n \geq 0$, $S(n) \geq n$ then it is fully space constructible.

If $f(n)$ and $g(n)$ are space constructible also $f(n)\, g(n)$, $2^{f(n)}$, and $f(n)\, 2^{g(n)}$ are space constructible. The functions log n, n, n^k, 2^n, and $n!$ are all fully space constructible.

We will assume analogous definitions of the *time constructible* functions and the *fully time constructible* functions. We have that n, n^k, 2^n, and $n!$ are all fully time constructible. Note that $\log n$ is *not* time constructible, because we assumed that time complexity is at least $n+1$, as indicated in Section 25 starting on page 113 and in [16, pages 299 and 316].

We have the following theorems.

Theorem 13. [Deterministic Space Hierarchy Theorem] If the function $S_2(n)$ is space-constructible, $inf_{n\to+\infty} S_1(n)/S_2(n) = 0$, $S_1(n) \geq \log_2 n$, and $S_2(n) \geq \log_2 n$, then there exists a language in DSPACE($S_2(n)$) which is *not* in DSPACE($S_1(n)$).

As a consequence of this theorem we have that:
(i) for any two real numbers r and s such that $0 \leq r \leq s$, DSPACE(n^r) $\subset$ DSPACE(n^s),
(ii) PSPACE does not collapse to DSPACE(n^k) for some $k \geq 0$, and
(iii) PSPACE $\subset$ EXPSPACE.

Theorem 14. [Deterministic Time Hierarchy Theorem] If the function $T_2(n)$ is fully time-constructible and $inf_{n\to+\infty} T_1(n) \log T_1(n)/T_2(n) = 0$, then there exists a language in DTIME($T_2(n)$) which is *not* in DTIME($T_1(n)$).

As a consequence of this theorem we have that:
P $\subset$ EXPTIME ($=_{def} \bigcup_{k\geq 0}$ DTIME(2^{n^k})) $\subset$ 2-EXPTIME ($=_{def} \bigcup_{k\geq 0}$ DTIME($2^{2^{n^k}}$)) $\subset$ and so on.

In particular, we have that there are problems solvable in DTIME(n^2), but not in DTIME(n), because $inf_{n\rightarrow+\infty}\, n\log n/n^2 = 0$, and analogously, there are problems solvable in DTIME(n^{53}), and not in DTIME(n^{52}). In general, we have that it does not exist a constant $max \geq 0$ such that for all $k \geq 0$, every problem solvable in DTIME(n^k) is solvable in DTIME(n^{max}). That is, P does *not* collapse to any deterministic polynomial time class DTIME(n^{max}), for some $max \geq 0$. If this were the case, then we would deduce that P $\neq$ PSPACE because we have that DTIME($f(n)$) $\subset$ DSPACE($f(n)$) (see Theorem 15 below).

The comparison of Theorems 13 and 14 tells us that with respect to the deterministic space hierarchy, in the deterministic time hierarchy we need to have a 'bigger distance' between two limiting functions for making sure that there exists a language recognized within the higher limiting function, but not within the lower one. This result shows that, in a sense, 'space' is a resource which is more valuable than 'time'.

29.2 Relationships among Complexity Classes

The following theorems establish relationships among classes of complexity measures [16, page 300 and 317] and [29, page 147]. For reasons of simplicity, in these theorems we assume that the function $f(n)$ is fully time constructible and fully space constructible.

Theorem 15. (i) DTIME($f(n)$) $\subseteq$ NTIME($f(n)$) $\subseteq$ DSPACE($f(n)$) $\subseteq$ NSPACE($f(n)$).
(ii.1) For every language L, if $L \in$ DSPACE($f(n)$) and $f(n) \geq \log_2 n$ then there exists $c > 1$ depending on L, such that $L \in$ DTIME($c^{f(n)}$).
(ii.2) For every language L, if $L \in$ NSPACE($f(n)$) and $f(n) \geq \log_2 n$ then there exists $c > 1$ depending on L, such that $L \in$ DTIME($c^{(\log n)+f(n)}$).
(iii) For every language L, if $L \in$ NTIME($f(n)$) then there exists $c > 1$ depending on L, such that $L \in$ DTIME($c^{f(n)}$).
(iv) DTIME($f(n)\,(\log f(n))$) $\subseteq$ DSPACE($f(n)$).

The weak inclusion NTIME($f(n)$) $\subseteq$ DSPACE($f(n)$) (see Point (i)), and Point (ii.2) are shown in [29, pages 147]. The other points are shown in [16, page 300 and 317].

Points (ii.1), (ii.2), and (iv) show again that 'space' is a resource which is more valuable than 'time'.

Obviously, the constant c occurring in the above Theorem 15, can be chosen to be 2.

Theorem 16. [Savitch's Theorem] If $L \in$ NSPACE($S(n)$), $S(n) \geq \log_2 n$, and $S(n)$ is fully space constructible then $L \in$ DSPACE($S^2(n)$).

We can informally rephrase the result of this theorem by saying that a nondeterministic Turing Machine which uses $S(n)$ space units has the same power of deterministic Turing Machine which uses $S^2(n)$ space units.

Note also that Savitch's Theorem leaves open the question of whether or not LIN-SPACE ($=_{def} \bigcup_{c>0}$DSPACE($c\,n$)) is equal to NLIN-SPACE ($=_{def} \bigcup_{c>0}$NSPACE($c\,n$)). (LIN stands for 'linear').

Savitch's Theorem only shows that NLIN-SPACE $\subseteq \bigcup_{c>0}$DSPACE$(c\,n^2)$. Since NLIN-SPACE is the class of languages accepted by linear bounded automata (which, by definition, are *nondeterministic* automata), the question of whether or not LIN-SPACE is equal to NLIN-SPACE is equivalent to the open question of whether or not deterministic linear bounded automata are equivalent to nondeterministic linear bounded ones.

29.3 Nondeterministic Space and Nondeterministic Time Hierarchies

The following two theorems are relative to nondeterministic polynomial space and polynomial time hierarchies. Note that the inclusions are proper inclusions.

Theorem 17. If $\varepsilon > 0$ and $r \geq 0$ then NSPACE$(n^r) \subset$ NSPACE$(n^{r+\varepsilon})$.
Theorem 18. If $\varepsilon > 0$ and $r \geq 0$ then NTIME$(n^r) \subset$ NTIME$(n^{r+\varepsilon})$.

As a consequence of this theorem we have that:
NP $\subset$ NEXPTIME $(=_{def} \bigcup_{k\geq 0}$ NTIME$(2^{n^k})) \subset$ 2-NEXPTIME $(=_{def} \bigcup_{k\geq 0}$ NTIME$(2^{2^{n^k}}))$ $\subset$ and so on.

29.4 Properties of General Complexity Measures

Theorem 19. [**Borodin's Gap Theorem**] Consider any total partial recursive function g such that $\forall n \geq 0\, g(n) \geq n$. Then there exists a total partial recursive function $S(n)$ such that DSPACE$(S(n))$ = DSPACE$(g(S(n)))$.

Theorem 20. [**Blum's Speed-up Theorem**] Let $r(n)$ be any total partial recursive function. There exists a recursive language L such that for all Turing Machine M_i accepting L in space $S_i(n)$, that is, $L \in$ DSPACE$(S_i(n))$, there exists a Turing Machine M_j accepting L in space $S_j(n)$ such that $r(S_j(n)) \leq S_i(n)$, for almost all n (that is, except a finite number of n's).

Theorems 19 and 20 hold also if we replace DSPACE by: (i) NSPACE (that is, we consider nondeterministic Turing Machines, instead of deterministic ones), or (ii) DTIME (that is, we consider, instead of the space, the time taken by a deterministic Turing Machine), or (iii) NTIME (that is, we consider, instead of the space, the time taken by a nondeterministic Turing Machine) [16, page 316].

As a consequence of Theorem 19 and its versions for deterministic and nondeterministic time and space, one can show the following unintuitive result: there exists a total partial recursive function $r(n)$ such that

DTIME$(r(n))$ = NTIME$(r(n))$ = DSPACE$(r(n))$ = NSPACE$(r(n))$.

As a consequence of Theorem 20 and its versions for deterministic and nondeterministic time and space, we have that there are functions which do not have 'best' programs (in time or space).

We also have the following result about Turing Machine computations (see [16, page 317]). Its proof is related to the way in which the spine functions of the Grzegorczyk Hierarchy grow (see Section 18 starting on page 81). Note also that the size of the input to a Turing Machine can be assumed to be a primitive recursive function of the input itself.

Theorem 21. A Turing Machine which can make a number of moves which is a primitive recursive function of the size of the input, can compute only a primitive recursive function of the input.

29.5 Computations with Oracles

In relation to the question of whether or not P is equal to NP we have the following two theorems which refer to the notion of a Turing Machine with oracles (see also [19, page 362]).

Let A be a language over the alphabet Σ. A Turing Machine with oracle A, denoted by M^A, is a usual 1-tape Turing Machine M with three extra states: q-ask, q-yes, and q-no. In the state q-ask the machine M asks the oracle A whether or not the string on its tape to the right of scanned cell until the leftmost blank, is in A. The answer is given in the next move by having the machine to enter the state q-yes or q-no. Then the computation continues normally until the state q-ask is entered again.

If A is a recursive language then for any Turing Machine M^A there exists an equivalent Turing Machine (without oracles). If A is not recursive then the language accepted by a Turing Machine M^A may be not recursively enumerable.

Let P^A be the set of languages accepted in polynomial time by a deterministic Turing Machine with oracle A. Let NP^A be the set of languages accepted in polynomial time by a nondeterministic Turing Machine with oracle A. We have the following result.

Theorem 22. $\mathrm{P}^A = \mathrm{NP}^A$, where A is the oracle which tells us in one time unit whether or not a quantified boolean formula is true.

This Theorem 22 holds also for any oracle A which solves a PSPACE-complete problem.

We also have the following result.

Theorem 23. There exists an oracle B such that $\mathrm{P}^B \neq \mathrm{NP}^B$.

30 Polynomial Hierarchy

Let us consider the computational complexity classes of problems depicted in Figure 34 on page 139. The set of problems which lies between the class P and the class PSPACE can be stratified according to the so called *Polynomial Hierarchy*, denoted PH. Note, however, that maybe not all problems between P and PSPACE can be stratified in the Polynomial Hierarchy because, as we will see below, it is an open problem whether or not PH is properly contained in PSPACE.

Within the Polynomial Hierarchy we have the classes indicated in the following Definition 10, where: (i) by P we denote the class of problems solvable by a deterministic Turing Machine in polynomial time (see Definition 1 on page 115 in Section 26), (ii) by C^{Oracle} we denote the class of all problems which are solvable in the class C by a Turing Machine which has an oracle in the class *Oracle*, (iii) we use the numerical subscripts such as $0, k$, and $k+1$, for indexing the various classes of problems within the Polynomial Hierarchy, and (iv) the superscript P is used to recall that the various classes are subclasses of the Polynomial Hierarchy.

Definition 10. [**Polynomial Hierarchy**] Let us stipulate that:

$$\Delta_0^{\mathrm{P}} = \Sigma_0^{\mathrm{P}} = \Pi_0^{\mathrm{P}} = \mathrm{P}, \quad \text{and}$$

for any $k \geq 0$, $\Delta_{k+1}^{\mathrm{P}} = \mathrm{P}^{\Sigma_k^{\mathrm{P}}}$, $\Sigma_{k+1}^{\mathrm{P}} = \mathrm{NP}^{\Sigma_k^{\mathrm{P}}}$, $\Pi_{k+1}^{\mathrm{P}} = \text{co-}\Sigma_{k+1}^{\mathrm{P}}$.

The *Polynomial Hierarchy* PH is defined to be: $\bigcup_{k \geq 0} \Sigma_k^{\mathrm{P}}$ (see also Figure 35).

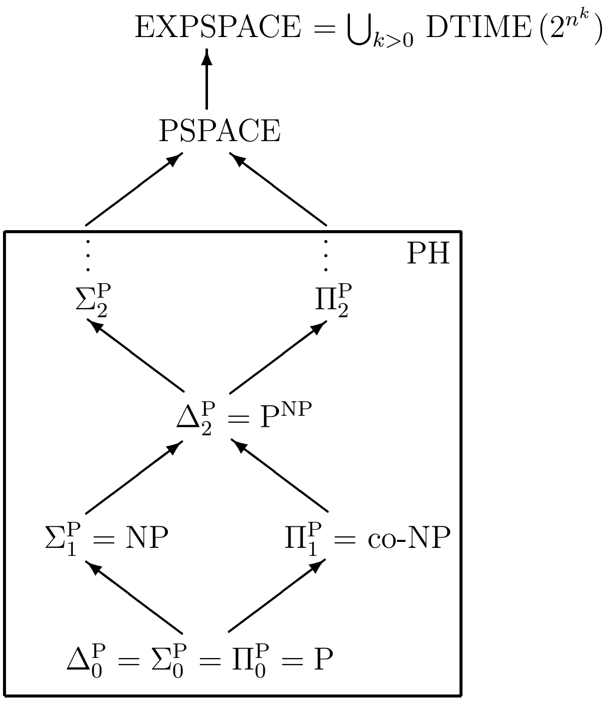

EXPSPACE = $\bigcup_{k>0}$ DTIME(2^{n^k})
PSPACE
PH
Σ_2^{P}
Π_2^{P}
$\Delta_2^{\mathrm{P}} = \mathrm{P}^{\mathrm{NP}}$
$\Sigma_1^{\mathrm{P}} = \mathrm{NP}$
$\Pi_1^{\mathrm{P}} = \text{co-NP}$
$\Delta_0^{\mathrm{P}} = \Sigma_0^{\mathrm{P}} = \Pi_0^{\mathrm{P}} = \mathrm{P}$

Fig. 35. The Polynomial Hierarchy PH. An arrow from A to B means $A \subseteq B$.

It is easy to see that:

(i) $\Pi_{k+1}^{\mathrm{P}} = (\text{co-NP})^{\Sigma_k^{\mathrm{P}}}$, and

(ii.1) $\Delta_1^{\mathrm{P}} = \mathrm{P}$, (ii.2) $\Sigma_1^{\mathrm{P}} = \mathrm{NP}$, (ii.3) $\Pi_1^{\mathrm{P}} = \text{co-NP}$.

We also have the following results which make PH to be a hierarchy.

(a) $\Delta_k^{\mathrm{P}} \subseteq \Sigma_k^{\mathrm{P}} \cap \Pi_k^{\mathrm{P}}$,

(b) $\Sigma_k^{\mathrm{P}} \cup \Pi_k^{\mathrm{P}} \subseteq \Delta_{k+1}^{\mathrm{P}}$.

It is an open problem whether or not in (a) or (b) above, $\subseteq$ is, in fact, a strict containment. In particular, for $k = 1$ we have that it is an open problem whether or not $\mathrm{P} \subset \mathrm{NP} \cap$ co-NP, as we already know.

We have that PH $\subseteq$ PSPACE. It is open whether or not PH $\subset$ PSPACE [19, page 98].

We also have that [19, page 97]:

(i) P = NP iff P = PH,

(ii) if P $\neq$ NP then PH $\subseteq$ ETIME, where the class ETIME, also called *single exponential time*, is defined as follows: ETIME $= \bigcup_{c>0}$DTIME(2^{cn}), and

(iii) If $\Sigma^{\mathrm{P}}_k = \Sigma^{\mathrm{P}}_{k+1}$ for some $k \geq 0$, then $\Sigma^{\mathrm{P}}_k = \mathrm{PH}$, that is, the Polynomial Hierarchy collapses at level k.

Now we present an interesting characterization of the sets of the Polynomial Hierarchy in terms of quantified formulas with alternate quantifiers. The reader will note the formal correspondence with the Arithmetical Hierarchy (see Section 24 on page 108).

Let us consider an alphabet A and the set A^* of words over A. For all $k \geq 0$, the set Σ^{P}_k is the set of all languages $L \subseteq A^*$ such that

$$L = \{x \mid \exists y_1 \forall y_2 \ldots Q y_k \, \langle x, y_1, y_2, \ldots, y_k\rangle \in R\}$$

where: (i) the quantifiers alternate (thus, Q is $\exists$ if k is odd, and Q is $\forall$ if k is even), and (ii) for all $i \in \{1, \ldots, k\}$, $|y_i| \leq p(|x|)$, where p is a polynomial, and (iii) the $(k{+}1)$-ary relation R can be recognized in polynomial time by a deterministic Turing Machine.

In particular, the set NP is the set of all languages L such that

$$L = \{x \mid \exists y \, \langle x, y\rangle \in R\}$$

where: (i) $|y| \leq p(|x|)$, where p is a polynomial, and (ii) the binary relation R can be recognized in polynomial time by a deterministic Turing Machine.

One can show that the set of true sentences of quantified boolean expressions with k alternate quantifications starting with $\exists$, is Σ^{P}_k-complete with respect to polynomial reductions. An instance of this Σ^{P}_2-complete problem is the following problem: we are given a formula φ constructed out of the operators *not*, *and*, *or*, and the variables y_1 and y_2, and we ask ourselves whether or not there exists a truth assignment to y_1 such that for all truth assignments to y_2 we have that φ is true.

The characterization of Π^{P}_k is obtained from the one of Σ^{P}_k by replacing $\exists$ with $\forall$ and vice versa. Thus, for all $k \geq 0$, the set Π^{P}_k is the set of all languages $L \subseteq A^*$ such that

$$L = \{x \mid \forall y_1 \exists y_2 \ldots Q y_k \, \langle x, y_1, y_2, \ldots, y_k\rangle \in R\}$$

where: (i) the quantifiers alternate (thus, Q is $\forall$ if k is odd, and Q is $\exists$ if k is even), and (ii) for all $i \in \{1, \ldots, k\}$, $|y_i| \leq p(|x|)$, where p is a polynomial, and (iii) the $(k{+}1)$-ary relation R can be recognized in polynomial time by a deterministic Turing Machine.

The set of true sentences of quantified boolean expressions with k alternate quantifications starting with $\forall$, is Π^{P}_k-complete with respect to polynomial reductions.

31 Invariance with respect to the Computational Model

In this section we will present an important fact, which explains why the complexity results obtained with reference to the Turing Machine model can be applied also to the case when one uses other models of computations like, for instance, the von Neumann machine [32, Chapter 2].

In particular, we will show that if a von Neumann machine takes n steps to solve a problem then there exists a Turing Machine which takes at most $p(n)$ steps to solve that problem, where $p(n)$ is a polynomial in n.

The proof of this fact is done in three steps: first, (i) we establish the polynomial relationship between the time complexities of a version of the Random Access Machine, called IA-RAM (see below), and the Turing Machine, based on the simulation presented in Section 8 on page 42, then (ii) we show that also the time complexities of the IA-RAM *without* the operations of multiplication and division, and the IA-RAM *with* the operations of multiplication and division are polynomially related (see Definition 11 on page 151), and finally, (iii) we prove that if we allow the program of a Random Access Machine to be modified while the computation progresses (see the SP-RAM model introduced below) and we disallow the indirect addressing, then we affect the time and space complexity measures, with respect to the case of the IA-RAM machine, only by a multiplicative factor.

Since for no polynomial $p(n)$ we have that: $O(p(n)) = O(2^n)$, and for no polynomial $p(n)$ we have that: $O(2^{p(n)}) = O(2^{2^n})$ all results concerning the polynomial (and exponential and double-exponential, etc.) time and space complexity of algorithms hold for Turing Machine as well as for IA-RAM's and SP-RAM's.

Let us start by introducing the two models of Random Access Machines we have mentioned above [2].

The first kind is the *Indirect Address* RAM, IA-RAM for short. This machine has one input read-only tape, one output write-only tape, a countable infinite number of memory locations (or registers), and a program counter (which is *not* a memory location). The first memory location is assumed to be the accumulator.

We assume that the registers and the accumulator may hold *arbitrary large integers*, and if we have to take into account the value of these integers then we have to consider also the size of their binary encodings.

We also assume that the program *cannot* be modified and it does not reside in memory (like the finite control of a Turing Machine which is *not* in the working tape).

The instructions available are the usual ones of a basic RAM [32, Chapter 2], that is, LOAD, STORE, ADD (addition), SUB (subtraction), MULT (multiplication), DIV (division), IN (read), OUT (write), JMP (jump), JP (jump on positive), JN (jump on negative), JZ (jump on zero), STOP. The *address part* of an instruction is either an integer denoting the register where the operand is stored, or it is a label where to jump to.

We assume that instructions may have *indirect addressing.* In this case we mark the address part by an asterisk (for instance, we write: LOAD $*$a).

The second kind of Random Access Machine is the *Stored Program* RAM, SP-RAM for short. It is like the IA-RAM, but there is no indirect addressing and it is possible to modify the program which resides in memory. (The IA-RAM is called RASP in [2].) The SP-RAM is the basic version of the von Neumann machine as described in [32, Chapter 2].

We may say that an IA-RAM is like a Turing Machine with the program encoded in the finite control, while an SP-RAM is like a Turing Machine with the program stored in the working tape. The von Neumann machine is a sort of SP-RAM (because the program can be changed), while the Turing Machine with the tape containing initially the input only (not the program, which is in the finite control), is a sort of IA-RAM, because the program cannot be changed.

Let N be the set of natural numbers, and let us consider the natural numbers $x_1,\ldots, x_n$. If (the encodings of) those numbers are on the leftmost n cells of the input tape, an IA-RAM (or an SP-RAM) that writes (the encodings of) the natural number y on the leftmost cell of the output tape, is said to compute the function $f(x_1, \ldots, x_n) = y$. It can be shown that the set of functions from N to N computed by an IA-RAM (or an SP-RAM) is exactly the set of the partial recursive functions from N to N.

Given a string s of symbols in the input tape, an IA-RAM (or an SP-RAM) that writes 1 on the leftmost cell of the output tape iff s belongs to a language L, is said to recognize the language L. It can be shown that the set of languages recognized by an IA-RAM (or an SP-RAM) is exactly the set of r.e. languages.

There are two criteria we may apply for measuring the space cost and time cost of the execution of an instruction of an IA-RAM or an SP-RAM. Let us first consider the case of an IA-RAM.

These two criteria are:
- the *uniform cost criterion* (or *constant cost criterion*): one register costs 1 unit of space and one instruction costs 1 unit of time, and
- the *logarithmic cost criterion*, where to store in a register the integer n costs $\lceil \log_2 |n| \rceil$ units of space (here $|n|$ denotes the absolute value of n and $\lceil x \rceil$ denotes the least integer which is greater than or equal to x) and the time units which are the cost for executing an instruction are indicated in Figure 36 on page 149.

Recall that we assume that the memory registers and the accumulator can hold arbitrarily large integers.

To explain the entries of Figure 36 we need to look at the operations involved during the execution of an instruction and, in particular, we need to take into account the cost of reading the address part of the instructions and the cost of reading and writing the integers in the memory registers or in the accumulator.

Instruction	Meaning	Cost (in units of time)	
1. LOAD a	$C(\mathrm{acc}) := C(\mathrm{a})$	$L(\mathrm{a})+L(C(\mathrm{a}))$	
LOAD $*$a	$C(\mathrm{acc}) := C(C(\mathrm{a}))$	$L(\mathrm{a})+L(C(\mathrm{a}))+L(C(C(\mathrm{a})))$	
2. STORE a	$C(\mathrm{a}) := C(\mathrm{acc})$	$L(C(\mathrm{acc}))+L(\mathrm{a})$	
STORE $*$a	$C(C(\mathrm{a})) := C(\mathrm{acc})$	$L(C(\mathrm{acc}))+L(\mathrm{a})+L(C(\mathrm{a}))$	
3. ADD a	$C(\mathrm{acc}) := C(\mathrm{acc})+C(\mathrm{a})$	$L(\mathrm{a})+L(C(\mathrm{a}))+L(C(\mathrm{acc}))$	(†)
ADD $*$a	$C(\mathrm{acc}) := C(\mathrm{acc})+C(C(\mathrm{a}))$	$L(\mathrm{a})+L(C(\mathrm{a}))+L(C(C(\mathrm{a})))+L(C(\mathrm{acc}))$	(†)
4. IN a	$C(\mathrm{a}) := \mathrm{input}$	$L(\mathrm{a})+L(\mathrm{input})$	
IN $*$a	$C(C(\mathrm{a})) := \mathrm{input}$	$L(\mathrm{a})+L(C(\mathrm{a}))+L(\mathrm{input})$	
5. OUT a	print $C(\mathrm{a})$	$L(\mathrm{a})+L(C(\mathrm{a}))$	
OUT $*$a	print $C(C(\mathrm{a}))$	$L(\mathrm{a})+L(C(\mathrm{a}))+L(C(C(\mathrm{a})))$	
6. JMP l	jump to label l	1	
7. JP l	jump to label if $C(\mathrm{acc}) > 0$	$L(C(\mathrm{acc})) + 1$	(††)
8. STOP	execution stops	1	

Fig. 36. The logarithmic costs (that is, the cost for the logarithmic cost criterion) of the instructions of the machine IA-RAM. (†): we do not indicate the analogous entries for SUB, MULT, and DIV. (††): we do not indicate the analogous entries for JN (with $C(\mathrm{acc}) < 0$) and JZ (with $C(\mathrm{acc}) = 0$). We may forget about the constant 1 because it is a term of lower order w.r.t. $L(C(\mathrm{acc}))$, whose order is $O(\log_2 C(\mathrm{acc}))$.

Given an integer n, let $L(n)$ denote its *binary length*, that is,

$L(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } \lceil \log_2 |n| \rceil$

and let $C(n)$ denote the integer contained in the memory register whose address is n.

As a first approximation, in the IA-RAM model we may assume that the cost t of executing an instruction like, for instance, 'LOAD a', is the cost of reading the address part a (proportional to $L(\mathrm{a})$) plus the cost of reading the operand (proportional $L(C(\mathrm{a}))$) and writing it into the accumulator (proportional $L(C(\mathrm{a}))$). We may assume that the proportionality ratios are 1, because in our cost analysis we are not actually interested in the multiplicative constants. Thus, we have that:

$t = L(\mathrm{a}) + L(C(\mathrm{a}))$.

If the instruction has indirect addressing then its logarithmic cost t (that is, its cost under the logarithmic cost criterion) is given by the following equation:

$t = L(\mathrm{a}) + L(C(\mathrm{a})) + L(C(C(\mathrm{a})))$.

The cost of fetching and decoding the operation code of an instruction is assumed to be constant in the IA-RAM model, because the program does not reside in memory and it is encoded in the finite control of the machine.

The content of the accumulator is denoted by $C(\mathrm{acc})$. We assume that the content of any register and the content of the accumulator can be made 0 in one unit of time.

Since for the IA-RAM the program cannot be changed and it is a finite sequence of instructions, we may assume that the labels of the instructions can be read in *constant* time. They can be stored in a *correspondence list* which is constructed before the execution of the program starts.

Example 4. If we consider the program for checking whether or not a string in $\{0,1\}^*$ of length n has the number of 0's equal to the number of 1's (that program uses a 'counter register' which goes one unit up when 1 is encountered and goes one unit down when 0 is encountered), then if we apply the uniform cost criterion, it takes $O(n)$ time (because the program has to scan the whole string) and $O(1)$ space (because the program uses one counter only).

If we apply the logarithmic cost criterion, it takes $O(n \log_2 n)$ time (indeed, the program performs at most n additions, and each partial sum has at most $\log_2 n$ bits because the value of a partial sum is at most n) and $O(\log_2 n)$ space (indeed, the program uses one register only and the content of that register is at most n). □

The space cost and the time cost of the execution of an instruction of an SP-RAM are like the ones for an IA-RAM for the uniform cost criterion, but in the case of the logarithmic cost criterion we have to take into account also:

- the time and space for accessing the content of the program counter, and
- the time and space spent during the 'fetch' phase and the 'decode' phase when interpreting an SP-RAM instruction. In this case, in fact, instructions may be changed and they reside in memory.

Let us now establish a polynomial relationship between the time complexity of the computations on the IA-RAM model and those on the deterministic Turing Machine model. We begin with the following definition.

Definition 11. **[Polynomially Related Functions]** Given two functions f and g, from N to N, they are said to be *polynomially related* iff there exist two polynomials p_1 and p_2 such that for all $n \in N$, we have that $f(n) \leq p_1(g(n))$ and $g(n) \leq p_2(f(n))$.

If we apply the uniform cost criterion, an IA-RAM can simulate in time $O(T(n))$ a k-tape Turing Machine which takes time $T(n)\,(\geq n)$ for any input of size n, because every cell of the tapes of the Turing Machine can be encoded by a register of the IA-RAM. Other registers of the IA-RAM are used to represent the positions of the tape-heads. The content of the tape cells of the Turing Machine can be accessed via indirect addressing through the memory registers containing the head positions.

If we apply the logarithmic cost criterion, the IA-RAM takes $O(T(n)\ \log_2(T(n)))$ to simulate the k-tape Turing Machine which takes time $T(n)\,(\geq n)$ for any input of size n. Indeed, for simulating p moves of the Turing Machine, the IA-RAM has to spend $(p \times \log_2 p)$ units of time, because every IA-RAM instruction takes logarithmic time to be executed (this time is proportional to the length of the address part of the instruction).

Thus, we have that an IA-RAM can simulate a k-tape Turing Machine which takes $T(n)$ time units, within $O(T^2(n))$ time units, if $T(n) \geq n$.

Let us now look at the converse simulation, that is, the one of a IA-RAM by a k-tape Turing Machine. Let us look at the following IA-RAM program for computing $x = 2^{(2^n)}$ for $n > 0$:

$i := 1$; $x := 2$; **while** $i \leq n$ **do begin** $x := x^2$; $i := i+1$ **end**.

To compute $x = 2^{(2^n)}$ this program takes $O(n)$ steps, while assuming that in each tape cell, the Turing Machine may write 0 or 1 only, it will take 2^n cells to store the result, thus it will take at least 2^n time units just to write the result.

Since n and 2^n are *not* polynomially related functions, for the uniform cost criterion the time complexity measures of Turing Machine operations are *not* polynomially related to those of an IA-RAM.

However, we have the following result.

Theorem 24. If an IA-RAM (without using multiplications or divisions) accepts a language L in time $T(n)\,(\geq n)$ for the logarithmic cost criterion, then a k-tape Turing Machine accepts L in time $O(T^2(n))$.

Proof. Let us recall the simulation of an IA-RAM by a k-tape Turing Machine (see Section 8 on page 42). We assume that the numbers in the registers of the IA-RAM are encoded in binary in the cells of the tapes of the Turing Machine.

Now we show that an IA-RAM computation of p time units takes at most $O(p^2)$ time units in a Turing Machine. Let us first observe that, after an IA-RAM computation of p time units, the non-blank portion of the memory tape of the Turing Machine is $O(p)$ long.

Let us now consider a 'STORE a' operation. It will take at most $O(p)$ steps in the Turing Machine, because it must search for the encoding of the cell 'a' where to store the content of the accumulator. Analogous time requirements can be established for the other IA-RAM operations (apart from the jump instructions). Thus, we have that an IA-RAM computation of p time units takes at most $O(p^2)$ time units in the Turing Machine. □

If we allow the IA-RAM to perform multiplications or divisions, each in one time unit, then p such operations take in the IA-RAM without multiplications or divisions $O(p^2)$ time units, because this last machine has to compute using additions (or subtractions) only. Actually, one can do better than $O(p^2)$.

As a conclusion, we have that the k-tape Turing Machine may require at most $O(T^4(n))$ time units to perform any computation done by an IA-RAM in $T(n)$ time units. Thus, their time complexities are polynomially related.

A theorem analogous to Theorem 24 holds for space measures, instead of the time measures.

We also have that the k-tape Turing Machine may require at most $O(T^8(n))$ time units, instead of $O(T^4(n))$ time units, to perform any computation done by an IA-RAM in $T(n)$ time units, if we insist that the Turing Machine should have one tape only. This is a consequence of the following result.

Theorem 25. A deterministic k-tape Turing Machine which computes a function in time $T(n)\,(\geq n)$ for any given input of size n, can be simulated by a one tape deterministic Turing Machine which computes the same function in time $(T(n))^2$.

Proof. Recall the quadratic slowdown (see Section 4.2 on page 26). □

We may conclude that to solve a particular problem a Turing Machine takes at least time (or space) $T(n)\,(\geq n)$ iff an IA-RAM takes at least $R(n)$ time (or space) units if we apply the logarithmic cost criterion, where $T(n)$ and R(n) are polynomially related.

Let us now prove that the time complexities of the SP-RAM model and the one of the IA-RAM model are related by a constant factor.

Theorem 26. (i) For the uniform cost criterion and the logarithmic cost criterion, for every IA-RAM program which takes $T(n)$ time there exists a constant k and an SP-RAM program which computes the same function and it takes at most $k \times T(n)$ time. (ii) The same holds by interchanging 'IA-RAM' and 'SP-RAM'.

Proof. (see also [2]) *Part* (i). It is enough to construct a program for an SP-RAM from any given program for an IA-RAM. All instructions can be left unchanged with the exception of the ones which make use of the indirect addressing. If an instruction has indirect addressing then we can use SP-RAM instructions:

(1) to save the content of the accumulator,

(2) to compute the address part of the corresponding instruction without indirect addressing,

(3) replace the old instruction (with indirect addressing) by the new instruction (without indirect addressing),

(4) to restore the content of the accumulator, and

(5) to execute the new instruction.

These operations require a constant number of SP-RAM instructions. Their total logarithmic cost is given by a linear combination of quantities which occur also in the expression of the logarithmic cost of the corresponding IA-RAM instruction. For instance, if we use the logarithmic cost criterion, the total time cost of the SP-RAM instructions for simulating the IA-RAM instruction 'ADD $*$a' via the actions (1)–(5) above, is:

$c_1\, L(\mathrm{a}) + c_2\, L(C(\mathrm{a})) + c_3\, L(C(C(\mathrm{a}))) + c_4\, L(C(\mathrm{acc}))$

Thus, for any given program of an IA-RAM there exists an equivalent program of an SP-RAM whose logarithmic time cost is bounded, within a constant factor, by the cost of the corresponding IA-RAM program. This constant factor depends on the value of the constants c_1, c_2, c_3, and c_4 relative to each IA-RAM instruction which may be subject to indirect addressing.

The same holds for uniform time cost criterion, instead of logarithmic time cost criterion.

Part (ii). Through an IA-RAM program we may simulate the process of *fetching*, *decoding*, and *executing* any SP-RAM instruction. This simulation can be done by an IA-RAM program which works as an interpreter (or a universal Turing Machine) by loading the given instruction, testing its operation code, incrementing the program counter, and performing the instruction on the IA-RAM (updating its memory, or its input tape, or its output tape, according to the operation code which has been found). If necessary, we will make the simulating IA-RAM to get the operand of the SP-RAM instruction to be simulated, by using instructions with indirect addressing.

It is not difficult to see that, in the case of the logarithmic cost criterion, the time needed for interpreting any given SP-RAM instruction is proportional (within a constant factor) to the time needed for executing the given SP-RAM instruction on a SP-RAM. This is basically due to the fact that the cost of interpreting any given SP-RAM instruction, say I, linearly depends on: (1) the number of the operations codes, (2) the length of its address part, and (3) the length of the operand. Since the number of operation codes is bounded by a constant, only the last two quantities affect the logarithmic cost of interpreting I, and they are already present in the expression of the logarithmic cost of executing I on a SP-RAM.

In particular, the interpretation of an SP-RAM instruction L which modifies a given SP-RAM instruction, can be done by changing the content of the register through which, by indirect addressing, the interpreting IA-RAM gets the operand. This interpretation process takes only a constant number of IA-RAM instructions, and their logarithmic cost is a linear combination (with constant factors) of quantities which occur also in the expression of the logarithmic cost of L on a SP-RAM. □

A theorem analogous to Theorem 26 holds for space measures, instead of time measures.

32 Complexity Classes Based on Randomization and Probability

This section will be based on notions we learnt from [19]. In order to introduce the classes of computational problems based on randomization and probability, we consider a particular class of nondeterministic Turing Machines with three tapes:

(i) an *input tape*, where the input is stored,

(ii) a *working tape*, where intermediate results are kept and modified, and

(iii) an *output tape*, where the output is printed.

Every computation performed by a nondeterministic Turing Machine for some input x, can be viewed as a tree of configurations called the *computation tree*: the root of that tree is the initial configuration with the code for x on the input tape, and each node of the tree has as children the configurations which can be reached by legal moves of the machine.

For every nondeterministic Turing Machine M we consider in this section, we assume, without loss of generality, that:
(i) M is used to solve decision problems and thus, its answers can only be 'yes' or 'no',
(ii) M accepts the input x iff in the computation tree for that input x there exists at least one configuration in which the machine enters a final state,
(iii) M stops when a final state is entered,
(iv) for every input x, the computation tree is *finite* and all the root-to-leaf paths of that tree have the same length.

The *time measure* of the computation of a Turing Machine M and an input x is defined to be the common length of all root-to-leaf paths of the computation tree T relative to M and x. The *space measure* is the maximum number, over *all* configurations of the computation tree T, of the cells of the working tape which are in use.

Without loss of generality, we make also the following assumptions on the nondeterministic Turing Machines we consider:
(v) all nodes in the computation tree have one or two sons, and
(vi) in every computation tree the root-to-leaf paths have the same number of *branch points*, that is, nodes with two sons.

Note that the hypothesis that a node may also have one son, is required by the fact that we want all paths to have the same length *and* the same number of branch points.

As a consequence of our assumptions (i)–(vi), we may associate with every leaf of a computation tree *either* (1) the answer 'yes' if in the configuration of that leaf the Turing Machine is in a final state, *or* (2) the answer 'no' if in the corresponding root-to-leaf path it does not exist any configuration where the Turing Machine is in a final state. Note that in our hypotheses, Case (1) and Case (2) are mutually exclusive and exhaustive.

Definition 12. [Random Turing Machines] A *random Turing Machine* is nondeterministic Turing Machine M such that for every input x, either (i) all leaves of the computation tree relative to M and x have answer 'no' or (ii) at least $\frac{1}{2}$ of the total number of the leaves have answer 'yes'.

A random Turing Machine can be simulated by a nondeterministic Turing Machine which at each branch point randomly chooses one of the two alternatives. Since we have assumed that all root-to-leaf paths in a computation tree have the same number of branch points, the probability that at a leaf we have the 'yes' answer is exactly the ratio: $\dfrac{\text{number of leaves with answer 'yes'}}{\text{total number of leaves}}$ (see also Figure 37 on page 155).

A random Turing Machine never gives the answer 'yes' when the answer is 'no', because if the answer is 'no' then all leaves must have the answer 'no'. However, when the answer is 'yes' a random Turing Machine gives the answer 'yes' with probability at least $\frac{1}{2}$.

This asymmetry with respect to the 'yes' and 'no' answer has the advantage that by repeating the simulation experiments, we can increase the level of confidence of the answers. In particular, if after n simulations we have obtained all answers 'no', then the probability that the correct answer is, instead, ‘yes’ is below $1/(2^n)$. Obviously, if we get the answer ‘yes’ in any of the simulation experiments, the correct answer is ‘yes’.

As a consequence of this increase of the level of confidence by repetition of the simulation experiments, in the above Definition 12, instead of $\frac{1}{2}$, we may equivalently consider any other number p such that $0<p<1$, with the condition that p should be independent of the input x. Obviously, if we choose a number p, instead of $\frac{1}{2}$, the random Turing Machine when the answer is ‘yes’, gives the answer ‘yes’ with probability at least p. However, with two simulation experiments that probability goes up to $2p - p^2$.

To see this, let us consider, for instance, the case where $p = \frac{1}{4}$. Then, by making two simulation experiments, instead of one only, we will have the answer ‘yes’ with probability $\frac{1}{4}\times\frac{1}{4} + 2\times(\frac{1}{4}\times\frac{3}{4}) = \frac{7}{16}$ (indeed, in the two experiments there is one case in which both answers are ‘yes’ with probability p^2, and there are two cases in which only one of the answers is ‘yes’ and each of these two cases has probability $p\times(1-p)$).

Definition 13. **[Class** RP] The class RP (also called R) is the class of decision problems which are solved by a random Turing Machine in polynomial time.

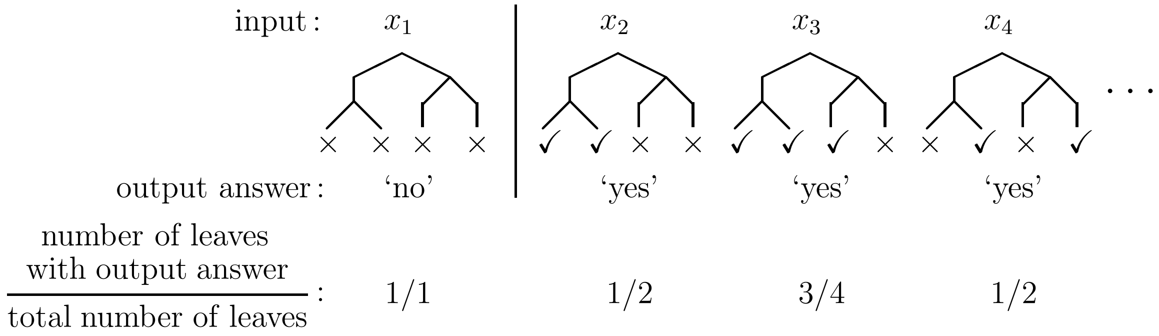


Fig. 37. Computation trees for a problem in RP and various inputs $x_1, x_2, \ldots$ In the leaves the symbol ✓ stands for the answer ‘yes’ and the symbol × stands for the answer ‘no’. The depth of the trees is a polynomial w.r.t. the size of the inputs x_i’s and it does not depend on the index i. We have assumed that: (i) all root-to-leaf paths have equal length, and (ii) every root-to-leaf path has the same number of branch points (indeed, every root-to-leaf path of the four trees of this figure has two branch points).

We have the following undecidability result [29, page 255].

Proposition 3. It does not exist an algorithm that
(i) always terminates, and
(ii) given a nondeterministic Turing Machine M, tells us whether or not M is in the class RP.

The following problem, which is *not* known to be in P (that is, the polynomial class), is in RP.

PRODUCT POLYNOMIAL INEQUIVALENCE

Input: Two sets $\{P_1, \ldots, P_n\}$ and $\{Q_1, \ldots, Q_m\}$ of polynomial with variables $x, y, \ldots$, over the rationals. Each polynomial is represented as the list of its terms with non-zero coefficients.

Output: 'yes' iff $\Pi_{1 \leq i \leq n} P_i$ and $\Pi_{1 \leq j \leq m} Q_j$ are different polynomials.

Note that the test whether or not $\Pi_{1 \leq i \leq n} P_i = \Pi_{1 \leq j \leq m} Q_j$ cannot be done by multiplying together the P_i's and the Q_j's, because the number of terms in the products can be exponential (w.r.t. the numbers of terms of the given polynomials). Likewise, we cannot factorize the P_i's and the Q_j's and check the resulting two lists of factors because also in this case, the number of factors can be exponential (w.r.t. the number of terms in the given polynomials). However, we can perform the test by using a random Turing Machine as follows. We choose in a random way a set of rational numbers where to evaluate each polynomial. Then, we multiply together the rational numbers resulting from the evaluation of those polynomials, and if their products are different then $\Pi_{1 \leq i \leq n} P_i \neq \Pi_{1 \leq j \leq m} Q_j$, and if they are equal then $\Pi_{1 \leq i \leq n} P_i = \Pi_{1 \leq j \leq m} Q_j$ with probability at least 0.5. Indeed, as it has been shown by Schwartz [35], if the set of rational points are chosen in a suitable random way, then in case $\Pi_{1 \leq i \leq n} P_i \neq \Pi_{1 \leq j \leq m} Q_j$, the products must differ half the time.

Definition 14. [Classes co-RP **and** ZPP**]** The class co-RP is the class of decision problems whose complement is in RP. The class ZPP (short for *zero probability* and *polynomial time*) is the class of decision problems in RP $\cap$ co-RP.

In the class co-RP the answer 'no' of the randomized Turing Machine is always correct, while the answer 'yes' may be incorrect and it is incorrect with probability smaller than 1/2. Note that in the class co-RP, with respect to the class RP, the roles of the answers 'yes' and 'no' are interchanged.

If a problem is in ZPP there exists a random Turing Machine which in polynomial time gives always the correct answer: it is enough to run both the random Turing Machine for RP and the random Turing Machine for co-RP in an interleaved way.

Due to this behaviour, every problem in ZPP is also said to have a polynomial 'Las Vegas' algorithm. This terminology refers to randomized algorithms which either give the correct answer or they do not give any answer at all. Generic randomized algorithms such as those of Definition 12 on page 154, are, instead, called 'Monte Carlo' algorithms.

Now let us consider a new class of nondeterministic Turing Machines which also satisfy the assumptions (i)–(vi) we have listed above. In this new class of nondeterministic Turing Machines, which is the class of probabilistic Turing Machines, the rules for giving the 'yes' or 'no' answers are different from those of random Turing Machines (see Definition 12 on page 154) and, in particular, there is a symmetry between the answer 'yes' and the answer 'no'.

Definition 15. [Probabilistic Turing Machines] A *probabilistic Turing Machine* is nondeterministic Turing Machine M such that for every input x, it gives the answer 'yes' if the computation tree for x has more than half of the leaves with answer 'yes', and it gives the answer 'no' if the computation tree for x has more than half of the leaves with answer ‘no’. If the computation tree for x has the same number of leaves with ‘yes’ and ‘no’ then the answer is ‘don’t know’.

Definition 16. [**Class** PP] The class PP is the class of all decision problems which are solved by a probabilistic Turing Machine in polynomial time.

The class PP contains NP ∪ co-NP and it is contained in PSPACE (see Figure 38 on page 157). We also have that the polynomial hierarchy PH is included in P^{PP}, that is, PH is included in the polynomial class P with oracles in PP [19, pages 119–120].

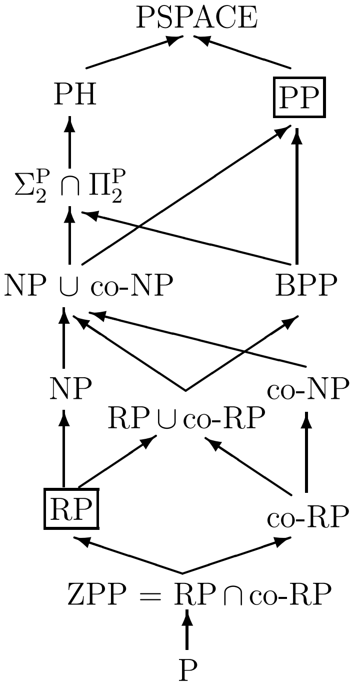


Fig. 38. Complexity classes for random (RP) and probabilistic (PP) Turing Machines with polynomial time complexity. An arrow from A to B means that $A \subseteq B$.

Unlike the problems in the class RP, by repeating the simulating experiments on a given problem of the PP class, we *cannot* increase the probability of the correctness of the answers.

However, the following definition identifies a subclass of the problems in PP in which we can increase the probability of the correctness of the answers by repeating the simulating experiments by using probabilistic, nondeterministic Turing Machines for solving the problems in that subclass.

Definition 17. [**Class** BPP] The class BPP is the class of all decision problems which are solved by a probabilistic Turing Machine in polynomial time in which the answer has the probability of $\frac{1}{2} + \delta$ of being correct, with $\delta > 0$.

The name BPP comes from PP and the B in front means that the probability is ‘bounded away from $\frac{1}{2}$’. Since the probability of the correctness of the answers can be increased by repeated simulating experiments, in the above Definition 17 we can replace $\frac{1}{2}+\delta$ (with $\delta>0$) by, for instance, $\frac{2}{3}$, without modifying the class BPP.

We have the following undecidability result [29, page 274].

Proposition 4. It does not exist an algorithm which (i) always terminates, and (ii) given a nondeterministic Turing Machine M, tells us whether or not M is in the class BPP.

We have that RP $\cup$ co-RP $\subseteq$ BPP $\subseteq$ PP (see Figure 38). We do not know any inclusion relationship between BPP and NP. Actually, if NP $\subseteq$ BPP then RP $=$ NP and the polynomial hierarchy collapses to BPP. However, we have that: BPP $\subseteq \Sigma_2^{\mathrm{P}} \cap \Pi_2^{\mathrm{P}}$.

33 Complexity Classes of Interactive Proofs

In this section we will consider the complexity classes which are related to interactive proof systems. We will follow the presentation done in [17].

In an interactive proof system there are two actors, a *prover,* also called Alice, and a *verifier*, also called Bob, and a statement to be proved which is initially known to both. As we will see, the notion of proof which is adopted in an interactive proof system, is *not* the one that is usually adopted in Mathematics, because the truth of the statement is established only within certain levels of probability.

The prover and the verifier have computing capabilities and they interact by exchanging messages. While computing, each actor has at his/her disposal a private sequence of random bits which is generated by a random source. Each message that is sent by an actor, is computed by taking into account the following information only: (i) the statement to be proved, (ii) the prefix of the random sequence used by the actor so far, (iii) the transcript of the messages exchanged so far, and (iv) new random bits from his/her private sequence.

At the end of the interaction, which can be announced by either actor, the verifier should decide with 'high probability' whether or not the given statement is true.

We can formalized an interactive proof system as follows (see [11]). There are two nondeterministic Turing Machines, called Alice and Bob, with a common input which is the statement x to be proved. Each machine has a private semi-infinite tape, called to *random tape*, which is filled with random bits. This tape is read from left to right, as more random bits are required. Alice and Bob send each other messages by writing them on a special *interaction tape* which can be read and written by both. Each machine is idle while the other is reading a new message from the interaction tape and computing the answer to that new message. Alice and Bob have also their own *input tape*, where x is stored, and their own *working tape.*

Alice has no computational limitations, except that every message she generates for Bob has its length polynomially bounded with respect to the size of x. Bob has computational limitations: the cumulative computation time at his disposal is polynomially bounded with respect to the size of x.

When either Alice or Bob has announced the end of the interaction, Bob has to accept or reject the truth of the given input statement x (maybe after making some more computation steps whose number is bounded by polynomial w.r.t. the size of x), 'with a certain probability level', as we now specify.

If we view an interactive proof system as a way of recognizing a language $L \subseteq \Sigma^*$ for some given alphabet Σ, that is, x is true iff $x \in L$, it should be the case that for any string $x \in \Sigma^*$, (i) (*completeness condition*) if $x \in L$ then Bob has to accept x with probability at least $\frac{2}{3}$, and (ii) (*soundness condition*) if $x \notin L$ then Bob has to reject x (that is, Bob tells that $x \notin L$) with probability at least $\frac{2}{3}$ (that is, Bob has to accept x with probability at most $\frac{1}{3}$).

Note that: (i) these probability values are related only to the sequence of random bits which are written in Bob's random tape and, thus, they do not depend on the sequence of random bits which are written in Alice's random tape, and (ii) we can replace the value $\frac{2}{3}$ by any other value which is bounded away from $\frac{1}{2}$ without changing the definition of the language L. Indeed, by repeating the execution of the interactive proof, we can reach any desired level of probability.

Definition 18. [Class IP] The class IP is the class of languages recognized by an interactive proof system.

We have that: NP $\subseteq$ IP. Indeed, every problem in NP can be solved by an interactive proof system as follows. Alice sends to Bob the encoding of the Turing Machine Alice and the list of the choices which are made by the nondeterministic Turing Machine Alice for accepting the input string x. Then Bob in polynomial time will act according to that list of choices and will accept x (with probability 1).

When, as in this case, Bob accepts with probability 1 (and not simply $\frac{2}{3}$) then the interactive proof system is said to satisfy the *perfect completeness condition* (not simply the completeness condition). It can be shown that every interactive proof system has an equivalent interactive proof system (that is, an interactive proof system which recognizes the same language) which satisfies the perfect completeness condition.

We also have that: BPP $\subseteq$ IP. Indeed, every problem in BPP can be solved by an interactive proof system as follows. Given a problem in BPP, Bob is the probabilistic Turing Machine whose existence ensures that the given problem is in BPP (recall that probabilistic Turing Machine are particular nondeterministic Turing Machines). Alice has nothing to do.

Now we show that the following problem is in IP.

GRAPH NON-ISOMORPHISM.
Input: two directed or undirected graphs G_1 and G_2.

Output: 'yes' iff the graphs G_1 and G_2 are non-isomorphic (that is, they are not equal modulo a permutation of the names of their nodes).

The interactive proof system which proves that the GRAPH NON-ISMORPHISM problem is in IP, is initialized by giving the two input graphs, say G_1 and G_2, both to Alice and Bob. The interactive proof has two rounds. The first round begins by Bob randomly choosing an i in $\{1, 2\}$. Then Bob sends to Alice a graph, say H, which is isomorphic to G_i (and this graph can be obtained by Bob by randomly permuting the vertices of G_i). Then Alice looks for a j in $\{1, 2\}$ such that graph G_j is isomorphic to H and she sends j back to Bob. (Note that Alice will always succeed in finding j, but if the two graphs G_1 and G_2 are isomorphic, j may be different from i.)

The second round is just like the first round.

When Bob receives the second value of j he announces the end of the interactive proof.

If the two given graphs are non-isomorphic, in both rounds Alice returns to Bob the same value he sent to her, and thus Bob accepts the given graphs as non-isomorphic with probability 1.

If the two given graphs are isomorphic, since in each round the probability that Alice guesses the i chosen by Bob is $\frac{1}{2}$, we have that the probability that Bob accepts the given graphs as non-isomorphic is at most $\frac{1}{4}$. That is, the probability that Bob tells that the two given graphs are isomorphic is at least $\frac{3}{4}\,(=1-\frac{1}{4})$, which is larger than $\frac{2}{3}$.

Note that we do not know whether or not the GRAPH NON-ISOMORPHISM problem is in NP. We only know that it is in co-NP, because the complementary problem, called the GRAPH ISOMORPHISM problem is in NP. The GRAPH ISOMORPHISM problem is defined as follows.

GRAPH ISOMORPHISM.
Input: two directed or undirected graphs G_1 and G_2.

Output: 'yes' iff the graphs G_1 and G_2 are isomorphic (that is, they are equal modulo a permutation of the names of their nodes).

We leave it to the reader to prove that the GRAPH ISOMORPHISM problem is in NP.

Note that it is an open problem whether or not the GRAPH ISOMORPHISM problem is NP-complete (see [10, page 285]).

We have the following result which we state without proof.

Proposition 5. IP = PSPACE.

We may extend the model of computation in interactive proof system by allowing $k\,(\geq 1)$ provers, instead of one only. This model of computation is said to be a *multi-prover interactive proof system.* As in the case of one prover, in a multi-prover interactive proof system the provers have no computational limitations, except for the fact that the messages they send to the verifier should be of polynomial length (w.r.t. the size of the statement to be proved). The verifier can communicate with each of the provers, but once the exchange of messages with the verifier has begun, the provers cannot communicate with each other. This restriction is due to the fact that, otherwise, the provers can make just one of them to do all the work without limiting the class of languages which the multi-prover interactive proof system can recognize [18]. The class of language which a multi-prover interactive proof system with $k\,(\geq 1)$ provers can recognize, is called MIP_k and it is defined as follows.

Definition 19. [Class MIP_k and MIP] For any $k \geq 1$, the class MIP_k is the class of languages recognized by a multi-prover interactive proof system with k provers. The class MIP of languages is defined to be $\bigcup_{k\geq 1} \mathrm{MIP}_k$.

Obviously, we have that: $\mathrm{IP} =_{def} \mathrm{MIP}_1 \subseteq \mathrm{MIP}_2 \subseteq \ldots \subseteq \mathrm{MIP}$. One can show the following results which we state without proof.

Proposition 6. (i) $\mathrm{MIP} = \mathrm{MIP}_2$. (ii) MIP = NEXPTIME.

Note that the result of this Proposition 6 is compatible with the fact that IP = MIP, because IP = PSPACE and we do not know whether or not PSPACE is equal to NEXPTIME (see Section 28.6 starting on page 137).

34 Complexity Classes of Parallel Computations

In this section we briefly present a few basic ideas and results on the complexity of parallel computations.

A model for parallel computation is the parallel random access machine, PRAM, for short. It is a machine with: (i) read-only input registers, (ii) write-only output registers, (iii) read-write registers which constitute a shared memory, and (iv) an *unbounded* number of processors. Often PRAM's are assumed to satisfy the so called 'concurrent-read, exclusive-write' restriction, CREW for short. This restriction means that a register can be read by two or more processors at the same time, but it cannot be written at the same time by more than one processor. All processors in a PRAM work in a synchronized way by performing exactly one step each, in parallel, at every time unit.

As an example of parallel computation, let us consider the following procedure for sorting an array A of N elements. For reasons of simplicity, we may assume that N is an integer power of 2.

```
parallel-sort(A):
  (i)   split A into halves, say A1 and A2,
  (ii)  parallel-sort(A1) || parallel-sort(A2), and
  (iii) merge the two arrays obtained at Step (ii) into a single array,
```

where $a \,||\, b$ means that the operations relative to the procedures (or processes) a and b are performed in parallel by two distinct processors. Having $N/2$ processors we can perform a tree-like computation for sorting, as depicted in Figure 39 on page 162. If we take the number of binary comparisons as the measure for the time complexity, we have the following upper bound on the time spent for merging:

1 (to compare 2 elements at level $(\log_2 N)-1$)
$+\,3$ (to merge two 2-element ordered lists at level $(\log_2 N)-2$)
$+\,7$ (to merge two 4-element ordered lists at level $(\log_2 N)-3$)
$\ldots$
$+\,(N-1)$ (to merge two $N/2$-element ordered lists at level 0).

The sum $1+3+7+\ldots+(N-1)$ is less than $2N$. Thus, by using $O(N)$ processors and $O(N)$ time units we can sort any given array A of size N.

Beside the PRAM model, various other models for parallel computations have been proposed, but if we allow binary comparison operations only, one can prove that all those models are polynomial-time interreducible, like in the deterministic sequential case we have that Turing Machine's and RAM's are polynomial-time interreducible.

Note, however, that if we allow concurrent reading and concurrent writing in registers, we may gain an exponential amount of time with respect to the case when reading and writing cannot be performed in a register by more than one processor at a time (see [13, page 273]).

However, the class of computable functions (from the natural numbers to the natural numbers) does not change when we allow parallel models of computation.

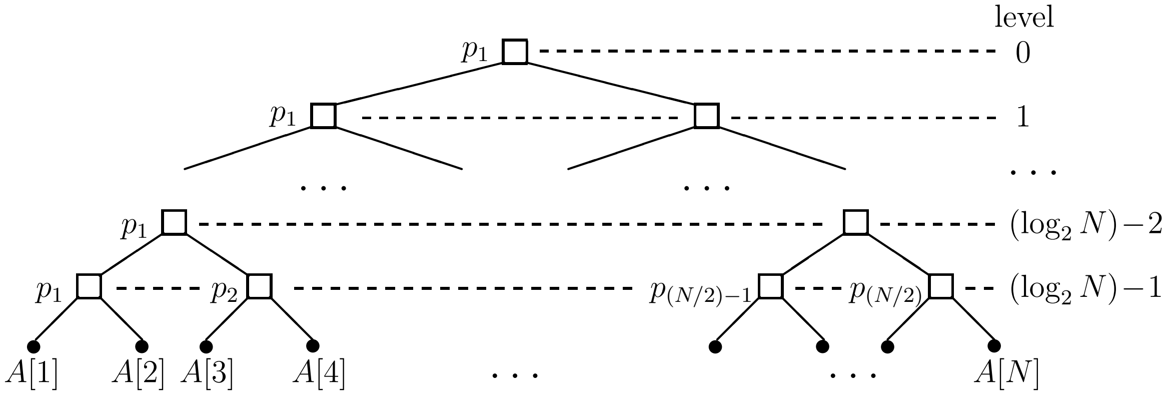


Fig. 39. Parallel sorting in a tree-like fashion of an array A of N elements: $A[1], \ldots, A[N]$. We have assumed that N is an integer power of 2.

If a problem can be solved in parallel time $T(n)$ (that is, time $T(n)$ using a PRAM model) then it can be solved in sequential space $p(T(n))$ (that is, space $p(T(n))$ using a sequential Turing Machine model (see Section 2)), where $p(m)$ is a polynomial in m.

If a problem can be solved in sequential space $S(n)$ then it can be solved in parallel time $p(S(n))$, where $p(m)$ is a polynomial in m (but we may need an exponential number of processors). Note, however, that this fact holds in a model of parallel computation where the time spent for communications among processors is *not* taken into account.

Thus, if $T(n)$ and $S(n)$ are polynomials, we get that:
polynomial space for sequential computations (that is, PSPACE) = polynomial time for parallel computations.

Remark 1. Since the question of whether or not PSPACE is equal to P is open, we cannot answer the question of whether or not using parallel models of computations, instead of the sequential ones, we actually increase the class of functions which can be computed in polynomial time.

Now let us introduce a class of problems which is of interest in case of parallel models of computations. It is called the NC class in honor of the researcher Nicholas Pippenger who studied it.

Definition 20. [Class NC] A problem is said to be in NC iff it is solvable on a PRAM which has a polynomial number of processors and runs within a poly-logarithmic time, that is, for input of size n, it runs in time $O((\log n)^k)$, for some $k \geq 1$.

Together with many other problems, sorting is a problem in NC. Indeed, there is an algorithm due to by R. Cole, which sorts on a 'CREW comparison PRAM' (that is, a CREW PRAM where assignments and binary comparison operations only are allowed) using for any input of size n, $O(n)$ processors, and $O(\log n)$ time, when counting also the time spent for: (i) deciding which comparisons should be performed on which processor, (ii) moving the elements within the various cells of the shared memory, and (iii) performing the binary comparisons.

For further details see, for instance, [20].

It can be shown that NC $\subseteq$ P. It is open whether or not the containment is proper.

Obviously, if some P-complete problem (with respect to polynomial time reductions) is in NC then NC $=$ P.

Index

References

1. M. Agrawal, N. Kayal, and N. Saxena. Primes is in P. Technical report, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur 208016, India, August 2002. http://www.cse.iitk.ac.in/news/primality.pdf.
2. A. Aho, J. E. Hopcroft, and J. D. Ullman. *Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
3. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, Volume 1. Prentice Hall, 1972.
4. P. Axt. Iteration of primitive recursion. *Zeit. für Mathematische Logik u. Grundlagen d. Math.*, 11:253–255, 1965. Also: Abstract 597-182, Notices Amer Math Soc, Jan. 1963.
5. H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, Amsterdam, 1984.
6. N. J. Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.
7. G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, N.J., 1963.
8. M. Davis. *Computability and Unsolvability.* McGraw-Hill, New York, 1958.
9. H. B. Enderton. *Elements of Set Theory.* Academic Press, New York, 1977.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* H. Freeman, San Francisco, USA, 1978.
11. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.
12. A. Grzegorczyk. Some classes of recursive functions. Technical report, Rozprawy Matematyczny IV, Instytut Matematyczny Polskiej Akademii Nauk, Warsaw, Poland, 1953.
13. D. Harel. *Algorithmics. The Spirit of Computing.* Addison Wesley, 1987.
14. M. A. Harrison. *Introduction to Formal Language Theory.* Addison Wesley, 1978.
15. H. Hermes. *Enumerability, Decidability, Computability.* Springer Verlag, 1969.
16. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
17. D. J. Johnson. The NP-Completeness column: An ongoing guide. *Journal of Algorithms*, 9:426–444, 1988.
18. D. J. Johnson. The NP-Completeness column: An ongoing guide. *Journal of Algorithms*, 13:502–524, 1992.
19. D. S. Johnson. A Catalog of Complexity Classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Volume A, pages 67–161. Elsevier, 1990.
20. R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Volume A, chapter 17, pages 869–941. Elsevier, 1990.
21. L. G. Khachyan. A polynomial algorithm in linear programming. *Doklady Akedamii Nauk SSSR*, 244:1093–1096, 1979. (in Russian). *U.S.S.R. Comput. Math. and Math. Phys.*, 20:53–72, 1980. (in English).
22. S. C. Kleene. *Introduction to Metamathematics.* North-Holland, 1971.

23. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.
24. H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.
25. Z. Manna. *Mathematical Theory of Computation.* MacGraw-Hill, 1974.
26. E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, California, Usa, Monterey, California, Usa, 1987. Third Edition.
27. A. R. Meyer. Weak monadic second order theory of successor is not elementary recursive. In R. Parikh, editor, *Proceedings Logic Colloquium (Symposium on Logic, Boston, Mass., 1972)*, Lecture Notes in Mathematics 453, pages 132–154, 1975.
28. A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967* 22*nd National Conference*, pages 465–469, New York, NY, USA, 1967. ACM Press.
29. C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.
30. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.
31. R. Péter. *Recursive Functions.* Academic Press, 3rd revised edition, 1967.
32. A. Pettorossi. *Quaderni di Informatica. Parte I.* Aracne, Second edition, 2004.
33. H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.
34. L. Sanchis. *Reflexive Structures: An Introduction to Computability Theory.* Springer Verlag, 1988.
35. J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27:710–717, 1980.
36. G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Proceedings ICALP '97*, Lecture Notes in Computer Science 1256, pages 671–681, 1997.
37. C. E. Shannon. A universal Turing machine with two internal states. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129–153. Princeton-University Press, Princeton, N.J., 1956.
38. S. Sippu and E. Soisalon-Soininen. *Theory of Parsing*, Volume 2. Springer-Verlag, 1990.
39. A. Turing. On computable numbers, with application to the Entscheidungsproblem. *Proc. London Mathematical Society. Series 2*, 42:230–265, 1936. Correction, ibidem, 43, 1936, 544–546.
40. K. Wagner and G. Wechsung. *Computational Complexity.* D. Reidel Publishing Co., 1990.