Computability and Complexity

Lecture Notes

Winter Semester 2019/2020

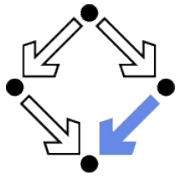
Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)

Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at

March 23, 2020



Contents

List of Definitions

List of Theorems

List of Theses

List of Figures

Notions, Notations, Translations

Abstraction Abstraktion
Acceptance problem Akzeptanz-Problem
Accepted akzeptiert
Accepting state akzeptierender Zustand
Ackermann function Ackermann-Funktion
Algorithm Algorithmus
Alphabet Alphabet
Ambiguity problem Zweideutigkeitsproblem
Amortized analysis amortisierte Analyse
Application Anwendung
Argument (Funktions-)Argument
Automaton (endlicher) Automat
Automaton language ($L(M)$) die Sprache eines Automaten
Average-case space complexity ($\overline{S}(n)$) durchschnittliche Raumkomplexität
Average-case time complexity ($\overline{T}(n)$) durchschnittliche Zeitkomplexität

Big-Ω notation ($\Omega(g(n))$) Omega-Notation
Big-O notation ($O(g(n))$) O-Notation
Bisimulation ($s_1 \sim s_2$) Bisimulation
Blank symbol ($\sqcup$) Leersymbol
Bottom ($\bot$) Bottom

Certificate Zertifikat
Characteristic function (1_P) charakteristische Funktion
Chomsky hierarchy Chomsky-Hierarchie
Closure properties Abschlusseigenschaften
Concatenation Verkettung
Configuration Konfiguration
Context-free kontextfrei

Context-sensitive kontextsensitiv
Cook's Theorem Satz von Cook
Countably infinite abzählbar unendlich

Decidable entscheidbar
Decision problem Entscheidungsproblem
Derivation ($w_1 \Rightarrow^* w_2$) Ableitung
Deterministic finite-state machine deterministischer endlicher Automat
DFSM deterministic finite-state machine
Difference equation Differenzengleichung
Direct derivation ($w_1 \Rightarrow w_2$) direkte Ableitung
Divide and conquer Teile und Herrsche
Domain ($domain(f)$) Definitionsbereich

Emptiness problem Leerheitsproblem
Empty word (ε) leeres Wort
Entscheidungsproblem Entscheidungsproblem
Enumerable aufzählbar
Enumerator Aufzähler
Extended transition function (δ^*) erweiterte Überführungsfunktion

Final state Endzustand
Finite closure (A^*) endlicher Abschluss
Finite language closure (L^*) endlicher (Sprach)abschluss
Fixed point operator (Y) Fixpunkt-Operatator
Function ($f : A \to B$) (totale) Funktion

Generated language ($Gen(M)$) erzeugte Sprache
Goto program Goto-Programm
Grammar Grammatik

Halting problem Halteproblem

Input Eingabe
Input alphabet (Σ) Eingabealphabet
Input symbol Eingabesymbol

Kleene's normal form Kleenesche Normalform

Lambda calculus (λ-calculus) Lambda-Kalkül
Lambda term (λ-term) Lambda-Term
Landau symbol Landau-Symbol
Language (L) Sprache
Language concatenation ($L_1 \circ L_2$) (Sprach)verkettung
Language equivalence Sprachäquivalenz
Language finiteness Endlichkeit einer Sprache
Language inclusion Spracheinschluss
Linear bounded automaton Linear beschränkter Automat
Little-ω notation ($\omega(g(n))$) Klein-Omega-Notation
Little-o notation ($o(g(n))$) Klein-O-Notation
Logarithmic cost model logarithmisches Kostenmodell
Loop computable Loop-berechenbar
Loop program Loop-Programm

Machine (Turing-)Maschine
Master theorem Hauptsatz der Laufzeitfunktionen
Minimization Minimierung
Move ($\vdash$) Zug
Mu-recursive (μ-recursive) My-rekursiv

NFSM nondeterministic finite-state machine
Nondeterministic finite-state machine nichtdeterministischer endlicher Automat
Nonterminal symbol Nonterminalsymbol
Non-trivial nicht-trivial
Normal form Normalform
NP-complete ($\mathcal{NP}$-complete) $\mathcal{NP}$-vollständig

Output tape Ausgabeband

Partial characteristic function ($1'_P$) partielle charakteristische Funktion
Partial function ($f : A \to_{\mathrm{p}} B$) partielle Funktion
Partitioning Zerlegung
Polynomial time verifier Polynomialzeit-Prüfer
Polynomial time-reducible Polynom-Zeit-reduzierbar
Post's Correspondence Problem Post'sches Korrespondenzproblem
Power set ($P(S)$) Potenzmenge

Primitive recursive primitiv recursiv
Problem (Entscheidungs)problem
Production rule Produktionsregel
Propositional formula aussagenlogische Formel
Pumping Lemma Pumping-Lemma (Schleifensatz)
Pumping length Aufpumplänge
Pushdown automaton Kellerautomat

Q-difference equation (q-difference equation) q-Differenzengleichung

RAM random access machine
Random access machine Registermaschine
Random access stored program machine Registermaschine mit gespeichertem Programm
Range ($range(f)$) Wertebereich
RASP random access stored program machine
Recognize erkennen
Recurrence (relation) Rekursionsgleichung
Recursive language rekursive Sprache
Recursively enumerable language rekursiv aufzählbare Sprache
Reducible ($P \leq P'$) reduzierbar
Regular expression ($Reg(\Sigma)$) regulärer Ausdruck
Regular expression language ($L(r)$) die Sprache eines regulären Ausdrucks
Regular language reguläre Sprache
Repetition (w^k) Wiederholung
Restricted halting problem eingeschränktes Halteproblem
Result ($f(a)$) (Funktions-)Ergebnis
Rice's Theorem Satz von Rice
Right linear rechtslinear

Satisfiability problem Erfüllbarkeitsproblem
Satisfiable erfüllbar
Semantics ($[\![P]\!]$) Semantik
Semi-decidable semi-entscheidbar
Sentence Satz
Sentential form Satzform
Space complexity ($S(n)$) Raumkomplexität
Stack Stapelspeicher

Start state (q_0) Anfangszustand
Start symbol Startsymbol
State Zustand
State equivalent ($s_1 \sim s_2$) (zustands)äquivalent
State partition ($[s]_p^S$) (Zustands)partition
State set (Q) Zustandsmenge
Subset construction Teilmengenkonstruktion
Symbol Symbol

Tape alphabet (Γ) Bandalphabet
Tape head Lese-/Schreibkopf
Tape symbol Bandsymbol
Term rewriting system Termersetzungssystem
Terminal symbol Terminalsymbol
Time complexity ($T(n)$) Zeitkomplexität
Transition function (δ) Überführungsfunktion
Turing complete Turing-vollständig
Turing computable (Turing-)berechenbar
Turing machine Turing-Maschine
Turing machine code ($\langle M \rangle$) Code einer Turing-Maschine
Turing machine language ($L(M)$) die Sprache einer Turing-Maschine

Universal universell
Universal language (L_u) universelle Sprache
Universal Turing machine (M_u) universelle Turing-Maschine

Verifier Prüfer

While computable While-berechenbar
While program While-Programm
Word Wort
Word concatenation ($w_1 \cdot w_2$) (Wort)verkettung
Word length ($|w|$) (Wort)länge
Word prefix ($w \downarrow k$) (Wort)präfix
Word problem Wortproblem
Working tape Arbeitsband
Worst-case space complexity ($S(n)$) Raumkomplexität im schlechtesten Fall
Worst-case time complexity ($T(n)$) Zeitkomplexität im schlechtesten Fall

Chapter 1.

Introduction

In these lecture notes we treat central parts of *theoretical computer science*, i.e., those aspects of computer science that represent "eternal truths". These truths have been established once and for all on the solid rock of mathematics and logic; they will not change any more by any kind of technological advances. In particular, we will deal with two core questions:

David Hilbert
(1862–1943)

- *computability:* based on formal models of computing, we will discuss what is computable and what is not;
- *complexity:* using precise models of computational costs, we will investigate how efficiently certain computing problems can be solved.

In the following, we will give a short historical account.

John von Neumann
(1903–1957)

Computability The question of computability has its roots in a time before the actual development of electronic computers. In the early 1920s, the influential mathematician David Hilbert suggested a work program for the development of a formal logical system for all of mathematics together with a proof that this system is consistent (no contradictory mathematical statements can be derived) and complete (all true mathematical statements can be derived). A mathematical statement would be thus true if and only if it could be formally derived in the system. In 1928 Hilbert also formulated the *Entscheidungsproblem*, namely to devise for such a system a mechanical procedure which can decide in a finite amount of time whether a given mathematical statement can be derived or not (i.e., whether the statement is true or not). Hilbert was convinced that a complete, consistent, and decidable formal logical system for mathematics could be found such that then mathematics could be reduced to mechanical computation.

Kurt Gödel
(1906–1978)

Hilbert's program seemed to be on a good way when John von Neumann proved in 1925 that first order predicate logic is consistent and Kurt Gödel proved in 1929 in his *completeness theorem* that this logic is also complete. However, it received a severe blow when in 1931

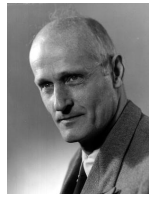
Kurt Gödel also proved in his first *incompleteness theorem* that arithmetic cannot be captured by any system that is both complete and consistent; furthermore he showed in his second incompleteness theorem that no sufficiently powerful consistent formal system can show its own consistency.

Alonzo Church
(1903–1995)

The program was ultimately destroyed, when in 1936 and 1937 Alonzo Church and Alan Turing independently showed that the Entscheidungsproblem for first order predicate logic could not be solved in two computational models they had devised for that purpose: the *lambda calculus* (Church) and a machine model (Turing) which was later called the *Turing machine*. Since Church and Turing could also show that their models had the same computational power, they conjectured that their models already covered any possible kind of computation (the *Church-Turing-Thesis*). Turing's proof was of particular influence, because it also showed that it can in general not be decided whether a Turing machine will eventually halt or not. Turing's machine model thus represented the first link between abstract mathematics/logic and the emerging field of computer science; furthermore, it already imposed some fundamental limits on this new field, five years before the first electronic computer was built!

Alan Turing
(1912–1954)

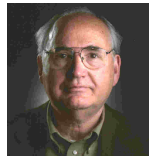
The field of theoretical computer science started to flourish. Based on prior work on the formal modeling of neural networks in the 1940s, in the 1950s numerous variants of the computational model of *finite state machine* (short: *automata*) were developed by Stephen Kleene, George Mealy, Edward Moore, and others. In 1959, Michael Rabin and Dana Scott clarified the field by introducing a simple automaton model that captured the essence of the idea; furthermore they introduced the influential concept of *nondeterminism* into automata theory (Rabin and Scott received for their work in 1976 the ACM Turing Award, the "Nobel prize" in computer science named in honor of Turing). Finite state machines have become one of the cornerstones of theoretical computer science, because (while being more restricted than Turing machines) they are still rich enough to describe many interesting kinds of applications and because many properties of finite state machines are decidable which are undecidable for Turing machines.



Stephen Kleene
(1909–1994)

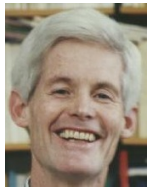
Michael Rabin (1931–)

Complexity Later on, it was more and more realized that solutions to computational problems must not only be *effective* (i.e., mechanically calculable) but also *efficient* (i.e., calculable with reasonable time and memory resources). The foundations of complexity theory were laid in the early 1960s by Alan Cobham and Jack Edmonds (who both popularized the complexity class $\mathcal{P}$ of problems solvable by deterministic Turing machines in polynomial time); the field was later influenced by the work of Juris Hartmanis and Richard Stearns (who received the ACM Turing Award in 1993). A core question of complexity theory (and still the most famous unsolved



Dana Scott (1932–)


Alan Cobham (1927–2011)


Stephen Cook (1939–)


Richard Karp (1935–)


Manindra Agrawal (1966–)

problem in theoretical computer science) became $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, i.e., whether nondeterministic Turing machines can solve more problems in polynomial time than deterministic ones can. The answer to this problem is of fundamental theoretical, practical, but also philosophical consequence: it would answer the question whether *constructing* a solution to a problem is indeed more complex than just *checking* whether a proposed solution is correct.

The question $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ led to the class of $\mathcal{NP}$-complete problems; if any of these problems could be solved deterministically in polynomial time, then all problems that can be solved nondeterministically in polynomial time could be also solved deterministically so. In 1971, Stephen Cook proved that there indeed exists an $\mathcal{NP}$-complete problem, the *satisfiability problem* in propositional logic; in 1972, Richard Karp showed that actually many more problems are $\mathcal{NP}$-complete (Cook received the Turing Award in 1982, Karp in 1985).

The area of complexity theory is still an active area of research. For instance, before 2002 many researchers conjectured (but could not prove) that there is no polynomial time algorithm which is able to decide whether a given number is prime or not. However, in 2002 exactly such an algorithm (the AKS primality test) was invented by Manindra Agrawal, Neeraj Kayal and Nitin Saxena. Also the newly emerging area of *quantum computing* imposes particular challenges to complexity theory; quantum computers can solve certain problems more efficiently than classical computers but the exact range of these problems is still unclear.

Lecture Notes The remainder of this document is structured in two parts according to the two core questions we consider. In Part I *Computability*, we first deal with the important but limited model of finite state machines and discuss its abilities and constraints; we then turn our attention to the more powerful model of Turing machines and equivalent computational models; finally we investigate the limits of computability by showing that not every computing problem is indeed solvable in these models. In Part II *Complexity*, we first investigate the notion of computational complexity; we will subsequently discuss the analysis of the complexity of computational methods and finally elaborate the border between practically feasible and infeasible computations in more detail.

These lecture notes are inspired by their predecessor [10] which was in turn based on [5]; however we have substantially revised the presentation and also added new material. An important source was [4] which contains a very readable German introduction into the field; also [8] and [9] were consulted. Our presentation of the complexity analysis of algorithm uses material from [2, 3, 7, 1, 6].

Acknowledgments Many thanks to Ralf Hemmecke and Burkhard Zimmermann for their detailed feedback on earlier drafts of these lecture notes that tremendously helped to clarify and correct the presentation. Should there be still any deficiencies, only the author is to blame.

Some Seminal Papers

- Alan M. Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society 2(42), p. 230–265, July–September, 1936.
- Alonzo Church. *An Unsolvable Problem of Elementary Number Theory*. American Journal of Mathematics 58(2), p. 345–263, 1936.
- Michael O. Rabin and Dana Scott. *Finite Automata and Their Decision Problems*. IBM Journal of Research and Development 3, p. 114–125, 1959.
- Juris Hartmanis and Richard E. Stearns *On the Computational Complexity of Algorithms*. Transactions of the American Mathematical Society 117, p. 285–306, 1965.
- Stephen A. Cook. *The Complexity of Theorem-Proving Procedures*. Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing, p. 151–158, ACM Press, New York, 1971.
- Stephen A. Cook and Robert A. Reckhow. *Time Bounded Random Access Machines*. Journal of Computer and System Sciences 7, p. 354–375, 1973.

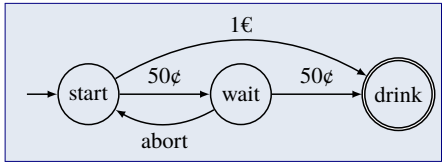
Part I.

Computability

Chapter 2.

Finite State Machines and Regular Languages

Suppose you want to model the behavior of a vending machine which operates as follows: the machine delivers a drink if you enter 1€, either by inserting a single 1€ coin or by inserting two 50¢ coins. After entering the first 50¢ coin, the machine does not allow you to enter 1€, because it cannot give change. However, you may abort the transaction by pressing a button; you will then receive your coin back. The behavior of this machine can be depicted by the following diagram (whose form will be explained later in more detail):



From this diagram you can read that there are infinitely many sequences of interactions that can persuade the machine to deliver a drink, e.g.

1€
50¢ 50¢
50¢ abort 1€
50¢ abort 50¢ abort 1€
50¢ abort 50¢ abort 50¢ 50¢
...

You might realize that all these sequences follow a common pattern: they end either with the single action "1€" or with the pair of actions "50¢ 50¢"; before that there may be arbitrarily

many repetitions of "50¢ abort". Such a sequence can be described in a more abstract form (which will be explained later in more detail) as

(50¢ abort)*(1€ + 50¢ 50¢)

You might wonder whether for every automaton of the kind shown above it is possible to describe in an analogous way the sequence of interactions leading to some form of "success".

In this chapter, we will formalize and answer this question: auto mata give rise to *finite-state machines* whose sequences of interactions can be described by *regular languages*. We will investigate the properties and the relationships of these two concepts.

2.1. Deterministic Finite State Machines

We start with an auxiliary definition.

Definition 1 (Alphabet) An *alphabet* is a finite set of elements called *symbols*.

Alphabet
Symbol

Then the core concept of this section is defined as follows.

Definition 2 (DFSM) A *deterministic finite-state machine* (short *DFSM*) M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with the following components:

- The *state set* Q, a finite set of elements called *states*.
- An *input alphabet* Σ, an alphabet whose elements we call *input symbols*.
- The *transition function* $\delta : Q \times \Sigma \to Q$, a function that takes a state and an input symbol as an argument and returns a state.
- The *start state* $q_0 \in Q$, one of the elements of Q.
- A set of *accepting states* (also called *final states*) $F \subseteq Q$, a subset of Q.

deterministischer endlicher Automat

Zustandsmenge
Zustand

Eingabealphabet
Eingabesymbol

Überführungsfunktion

Anfangszustand

akzeptierender Zustand

Endzustand

In the following, we will denote a DFSM simply by the term *automaton*.

(endlicher) Automat

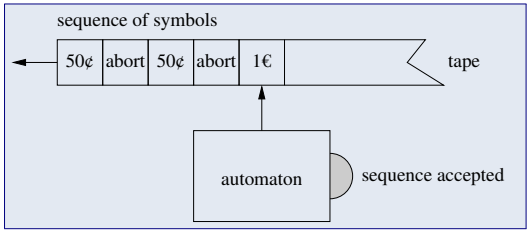


Figure 2.1.: A Deterministic Finite-State Machine

The behavior of such an automaton is illustrated by Figure 2.1 (which uses as its symbol set the set of interactions $\{50¢, 1€, \text{abort}\}$ of the vending machine described at the beginning of the chapter). The automaton operates on an input tape that contains a finite sequence of input symbols. At every time the automaton is in a certain state; every execution step of the automaton consists of reading one symbol from the tape by which the tape is moved one position to the left. Based on the current state and its input symbol, the transition function δ determines the next state of the automaton in which the next symbol is read. When all symbols have been read, the automaton indicates by a signal whether it is in an accepting state, i.e., whether it accepts the sequence. The essence of above definition is therefore the transition function δ which defines the behavior of M: if $\delta(q, x) = q'$, then this means that, if M is in state q and reads symbol x, it performs a transition into state q'.

The tape can be interpreted as an *input medium* whose content (the symbol sequence) is first fully constructed and then given to the automaton. One might however also construct this input *step by step* during the execution of the automaton, i.e., after the execution of every step the automaton waits for the the next input to arrive; this corresponds to a sequence of "on the fly" interactions with the automaton as described at the beginning of this chapter. One might even think of the tape as an *output medium* where the automaton announces to the environment its readiness to engage in a particular interaction (which then the environment has to accept to make progress). The selection of an appropriate interpretation of the model depends on the application context; the mathematical formalism remains the same for every interpretation.

The transition function is usually defined by a table with a line for every state q and a column for every symbol x such that the corresponding table entry denotes $\delta(q, x)$:

$M = (Q, \Sigma, \delta, q_0, F)$
$Q = \{q_0, q_a, q_r\}$
$\Sigma = \{\text{a}, \text{b}, 0, 1, ?\}$
$F = \{q_a\}$

δ	a	b	0	1	?
q_0	q_a	q_a	q_r	q_r	q_r
q_a	q_a	q_a	q_a	q_a	q_r
q_r	q_r	q_r	q_r	q_r	q_r

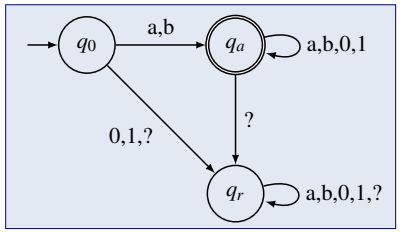


Figure 2.2.: A Deterministic Finite-State Machine

δ	$\ldots$	x	$\ldots$
$\vdots$			
q		$\delta(q, x)$	
$\vdots$			

The transition function can be also depicted by a directed graph (see Figure 2.2):

- Every state is depicted by a vertex.
- For every transition $\delta(q, x) = q'$, an arc with label x is drawn from state q to state q'. If there are multiple transitions from q to q', a single arc is drawn with multiple labels.
- The start state is marked by a small incoming arrow.
- Every accepting state is marked by a double outline.

Definition 3 (Word, Concatenation, Closure, Language)

- A *word* w over an alphabet A is a finite (possibly empty) sequence $w = a_1 a_2 \ldots a_n$ of symbols a_i from A (for some $n \geq 0$ and all $1 \leq i \leq n$). Wort
- We denote by $|w|$ the *word length* n. (Wort)länge
- We denote, for $0 \leq k \leq |w|$, by $w \downarrow k$ the *word prefix* $a_1 a_2 \ldots a_k$ of w. (Wort)präfix

leeres Wort
(Wort)verkettung
Verkettung
Wiederholung
endlicher Abschluss
Sprache

- The *empty word* is denoted by ε.
- By $w_1 \cdot w_2$ (or simply $w_1 w_2$) we denote the *word concatenation* (short *concatenation*) of word $w_1 = a_1 a_2 \dots a_n$ and word $w_2 = b_1 b_2 \dots b_m$ to the word $w_1 \cdot w_2 = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$.
- For every $k \geq 0$, the *repetition* w^k denotes the k-fold concatenation $w \cdot w \cdot \dots \cdot w$ (including the special cases $w^0 = \varepsilon$ and $w^1 = w$).
- The *finite closure* A^* is the set of all words over A.
- Every subset $L \subseteq A^*$ is called a *language* over A.

erweiterte Überführungsfunktion

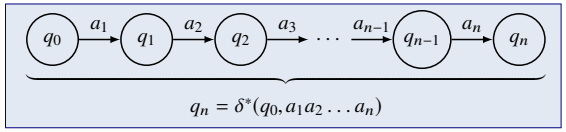
Definition 4 (Extended transition function) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFSM. We define the *extended transition function* $\delta^* : Q \times \Sigma^* \to Q$ of M, which takes a state and a *word* as an argument and returns a state as a result, inductively over the structure of its argument word. In more detail, for all $q \in Q, w \in \Sigma^*, a \in \Sigma$, we define δ^* by the two equations

$$\delta^*(q, \varepsilon) := q$$
$$\delta^*(q, wa) := \delta(\delta^*(q, w), a)$$

If, for some states q and q' and word $a_1 a_2 \dots a_n$, we have $q' = \delta^*(q, a_1 a_2 \dots a_n)$, then there exists a sequence of states $q = q_0, q_1, \dots, q_n = q'$ such that

$$q_1 = \delta(q_0, a_1)$$
$$q_2 = \delta(q_1, a_2)$$
$$\dots$$
$$q_n = \delta(q_{n-1}, a_n)$$

which we can depict as



In other words, the word $a_1 a_2 \dots a_n$ drives automaton M from state q to state q'.

die Sprache eines Automaten
akzeptiert

Definition 5 (Language of a DFSM) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFSM. Then the *automaton language* $L(M) \subseteq \Sigma^*$ of M is the language over Σ defined by

$$L(M) := \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

In other words, $L(M)$ is the set of all words that drive M from its start state to some accepting state. Word w is *accepted* by M, if $w \in L(M)$.

Example 1 (Language of Identifiers) The automaton in Figure 2.2 accepts all words that start with 'a' or 'b' and are followed by an arbitrary sequence of 'a', 'b', '0', '1', i.e., all words that consist of letters or digits but start with a letter. The automaton thus accepts

a, b, ab, a1, a1b, a101

but not the words "1a" or "a1?". Taking a bigger alphabet of letters and digits, the language of this automaton is the language of *identifiers* in many programming languages. □

Example 2 (Even Numbers) Take the program depicted in Figure 2.3 which reads symbols from an input stream and returns true if the stream contains an even number of '0' and an even number of '1' (and no other symbol). The program operates with two Boolean variables e_0 and e_1 that have value "true" if the number of '0' respectively '1' read so far is an even number. The variables are initialized to "true" (0 is an even number); when a symbol is read, the corresponding variable is negated (an even number becomes an odd number and an odd number becomes an even number). The program returns true only if the only symbols encountered are '0' and '1' and if both variables are "true".

The behavior of this program can be modeled by an automaton M, as depicted in Figure 2.3. The automaton has four states corresponding to the four possible combination of values of the program variables e_0 and e_1:

	e_0	e_1
q_0	true	true
q_1	true	false
q_2	false	true
q_2	false	false

```
function EvenZerosAndOnes()
    e0, e1 ← true, true
    while input stream is not empty do
        read input
        case input of
            0: e0 ← ¬e0
            1: e1 ← ¬e1
            default: return false
        end case
    end while
    return e0 ∧ e1
end function
```

$M = (Q, \Sigma, \delta, q_0, F)$
$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{0, 1\}$
$F = \{q_0\}$

δ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

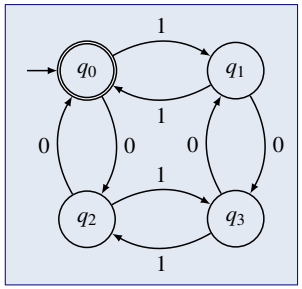


Figure 2.3.: Even Number of Zeros and Ones

The start state corresponds to the state of e_0 and e_1 at the start of the loop. This state is also the only accepting one, which corresponds to the values that e_0 and e_1 must have such that the program may return "true". Every transition of M corresponds to the negation of one of the variables in the body of the loop. □

2.2. Nondeterministic Finite State Machines

If a DFSM is in a state q, the next symbol x on the input tape uniquely determines the successor state $q' = \delta(q, x)$. We are now going to relax this condition such that for given q and x, there is more than one successor state q' (but also none) possible. Furthermore, we are going to allow the automaton to start in one of several possible start states.

We start with an auxiliary definition.

Definition 6 (Powerset) The *power set* $P(S)$ is the set of all subsets of set S:

$$P(S) := \{T \mid T \subseteq S\}$$

Potenzmenge

Then the core concept of this section is defined as follows.

Definition 7 (NFSM) A *nondeterministic finite-state machine* (*NFSM*) is a 5-tuple $M = (Q, \Sigma, \delta, S, F)$ with the following components:

nichtdeterministischer endlicher Automat

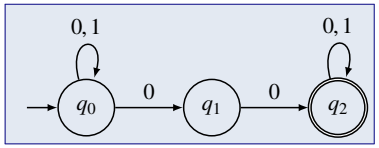
- The *state set* Q, a finite set of elements called *states*. — Zustandsmenge, Zustand
- An *input alphabet* Σ, an alphabet whose elements we call *input symbols*. — Eingabealphabet, Eingabesymbol
- The *transition function* $\delta : Q \times \Sigma \to P(Q)$, a function that takes a state and an input symbol as an argument and returns a set of states. — Überführungsfunktion
- A set of *start states* $S \subseteq Q$, a subset of Q. — Anfangszustand
- A set of *accepting states* (also called *final states*) $F \subseteq Q$, a subset of Q. — akzeptierender Zustand, Endzustand

According to above definition, for given state q and symbol x, a NFSM may have no, exactly one, or more than one successor state q', depending on whether $\delta(q, x)$ is empty, has exactly one element, or has multiple elements. In the following, we will use the term *automaton* to denote both DFSMs and NFSMs (the exact meaning should be clear from the context).

We can define a NFSM by a table or a graph as depicted in Figure 2.4.

An informal interpretation of a NFSM is that, when it is in a state q and reads a symbol x, it splits itself into multiple copies that *simultaneously* investigate all successor states in $\delta(q, x)$; an input word is accepted if there is at least one copy that reaches an accepting state.

Example 3 (Nondeterminism) Take the following NFSM over alphabet $\{0, 1\}$:



$M = (Q, \Sigma, \delta, S, F)$
$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$\Sigma = \{0, 1\}$
$S = \{q_0\}$
$F = \{q_2, q_4\}$

δ	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

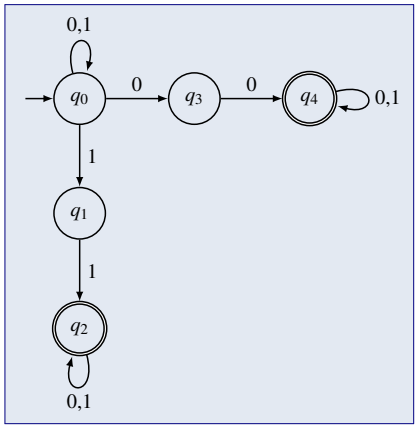
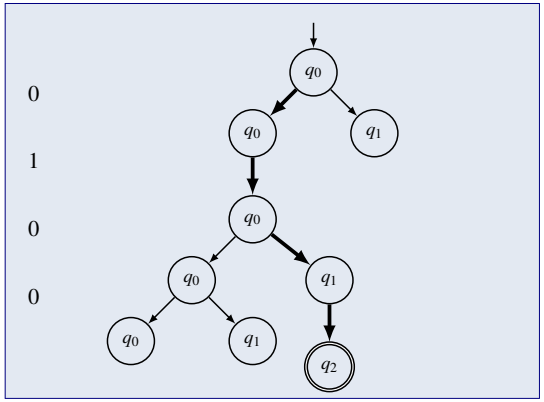


Figure 2.4.: A Nondeterministic Finite-State Machine

The nondeterministic behavior of the automaton on input 0100 can be visualized as follows:



In state q_0 on input 0, there exist transitions to q_0 and q_1; consequently the automaton starts to

investigate both possibilities in parallel. For the first occurrence of 0, the transition to q_1 leads to a dead end; however for the second occurrence of 0, this transition is the right choice which ultimately leads to the acceptance state q_2. □

Definition 8 (Extended transition function) Let $M = (Q, \Sigma, \delta, S, F)$ be a NFSM. We define the *extended transition function* $\delta^* : Q \times \Sigma^* \to P(Q)$ of M, which takes a state and a *word* as an argument and returns a set of states as a result, inductively over the structure of its argument word. In more detail, for all $q \in Q, w \in \Sigma^*, a \in \Sigma$, we define δ^* by the two equations

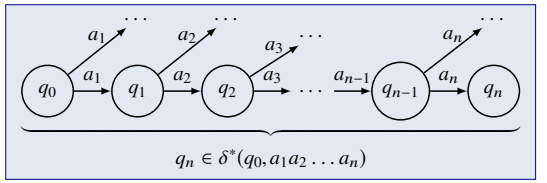
erweiterte Überführungsfunktion

$$\delta^*(q, \varepsilon) := \{q\}$$
$$\delta^*(q, wa) := \{q'' \mid \exists q' \in \delta^*(q, w) : q'' \in \delta(q', a)\}$$

If, for some states q and q' and word $a_1 a_2 \ldots a_n$, we have $q' \in \delta^*(q, a_1 a_2 \ldots a_n)$, then there exists a sequence of states $q = q_0, q_1, \ldots, q_n = q'$ such that

$$q_1 \in \delta(q_0, a_1)$$
$$q_2 \in \delta(q_1, a_2)$$
$$\ldots$$
$$q_n \in \delta(q_{n-1}, a_n)$$

which we can depict as



In other words, the word $a_1 a_2 \ldots a_n$ drives automaton M from state q to state q' (but potentially also to other states).

Definition 9 (Language and Acceptance of a NFSM) Let $M = (Q, \Sigma, \delta, S, F)$ be a NFSM. Then the *automaton language* $L(M) \subseteq \Sigma^*$ of M is the language over Σ defined by

die Sprache eines Automaten

$$L(M) = \{w \in \Sigma^* \mid \exists q \in S : \delta^*(q, w) \cap F \neq \emptyset\}$$

akzeptiert

Word w is *accepted* by M if $w \in L(M)$.

Thus a NFSM accepts a word if there exists some path from some start node to some accepting node such that the edges of the path are labeled by the symbols of the word in the corresponding sequence.

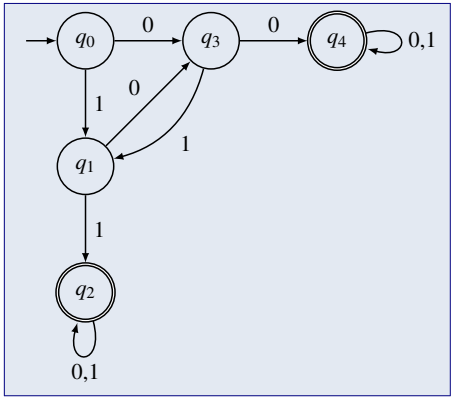
As can be seen by the following example, it is often easier to give a NFSM that accepts a certain language than to give a corresponding DFSM.

Example 4 (Finding Subwords) It is easy to see that the NFSM in Figure 2.4 accepts all words in which the substring 00 or the substring 11 occurs. All words with a substring 00 are recognized via a state sequence that includes state q_3, all words with a substring 11 are recognized via a state sequence that includes state q_1.

A word may be recognized by multiple state sequences, e.g. 00111 is recognized by any of the following sequences:

$$q_0 \xrightarrow{0} q_3 \xrightarrow{0} q_4 \xrightarrow{1} q_4 \xrightarrow{1} q_4 \xrightarrow{1} q_4$$
$$q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_2$$
$$q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2$$

It is harder to see that the DFSM with the following graph accepts the same language:



The idea behind this deterministic automaton is that q_1 is reached after a 1 has been recognized and q_3 is reached after a 0 has been recognized; a following symbol of the same kind drives the automaton to an accepting state while a following symbol of the other kind drives the automaton from q_1 to q_3 respectively from q_3 to q_1. □

Clearly, a DFSM is just a special case of a NFSM, thus for every language L that is accepted by some DFSM, there exists also a NFSM M' that accepts L. The following theorem proved by Rabin and Scott in 1959 tells us that (somewhat surprisingly) also the converse is true.

Teilmengenkonstruktion

Theorem 1 (*Subset construction*) Let $M = (Q, \Sigma, \delta, S, F)$ be a NFSM and let $M' = (Q', \Sigma, \delta', q'_0, F')$ be the following DFSM:

$$Q' = P(Q)$$
$$\delta'(q', a) = \bigcup_{q \in q'} \delta(q, a)$$
$$q'_0 = S$$
$$F' = \{q' \in Q' \mid q' \cap F \neq \emptyset\}$$

Then $L(M') = L(M)$.

This theorem states that nondeterminism does not really add anything to the expressive power of an automaton: for every NFSM M a corresponding DFSM M' with the same language can be constructed as follows: the states of M' are *sets* of states of M, the start state of M' is the set of start states of M, the accepting states of M' are those sets that contain accepting states in M; there exists in M' a transition from a state q'_1 to a state q'_2 if there exists in M a transition from some state $q_1 \in q'_1$ to some state $q_2 \in q'_2$.

Proof We show, for every word $w = a_1 a_2 \ldots a_n$, that $w \in L(M)$ if and only if $w \in L(M')$:

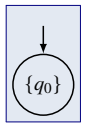
⇒ Assume $w = a_1 a_2 \ldots a_n \in L(M)$. Then there exists a state sequence $q_0, q_1, q_2, \ldots, q_n$ with $q_0 \in S$ and $q_n \in F$ and $q_1 \in \delta(q_0, a_1), q_2 \in \delta(q_1, a_2), \ldots, q_n \in \delta(q_{n-1}, a_n)$. Take the sequence of state sets $Q_0, Q_1, Q_2, \ldots, Q_n$ with $Q_0 = S$, $Q_1 = \delta'(Q_0, a_1), Q_2 = \delta'(Q_1, a_2)$, $\ldots, Q_n = \delta'(Q_{n-1}, a_n)$. We know $q_0 \in S = Q_0$; according to the definition of δ', we thus have $q_1 \in \delta(q_0, a_1) \subseteq \delta'(Q_0, a_1) = Q_1$; we thus have $q_2 \in \delta(q_1, a_2) \subseteq \delta'(Q_1, a_2) = Q_2$; $\ldots$; we thus have $q_n \in \delta(q_{n-1}, a_n) \subseteq \delta'(Q_{n-1}, a_n) = Q_n$. Since $q_n \in Q_n$ and $q_n \in F$, we have $Q_n \cap F \neq \emptyset$ and thus $w \in L(M')$.

⇐ Assume $w = a_1 a_2 \ldots a_n \in L(M')$. Then there exists a sequence $Q_0, Q_1, Q_2, \ldots, Q_n$ with $Q_0 = S$ and $Q_n \cap F \neq \emptyset$ and $Q_1 = \delta'(Q_0, a_1), Q_2 = \delta'(Q_1, a_2), \ldots, Q_n = \delta(Q_{n-1}, a_n)$. According to the definition of δ', there exists a sequence $q_0, q_1, q_2, \ldots, q_n$ with $q_0 \in Q_0, q_1 \in$

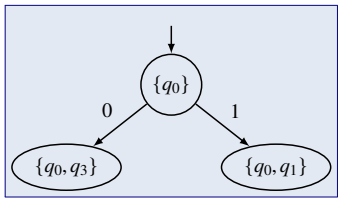
$Q_1, q_2 \in Q_2, \ldots, Q_n \in Q_n$ with $q_1 \in \delta(q_0, a_1), q_2 \in \delta(q_1, a_2), \ldots, q_n \in \delta(q_{n-1}, a_n)$ and $q_n \in F$. Since $q_0 \in Q_0 = S$, we have $w \in L(M)$. □

Since DFSMs and NFSMs have the same expressive power, our sloppy terminology to call both plainly *automata* is justified.

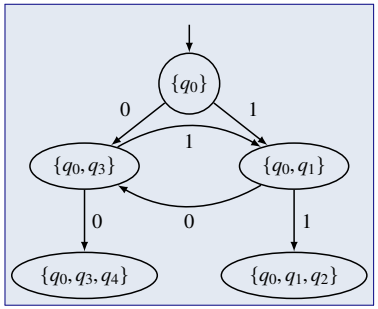
Example 5 (Subset construction) We apply the subset construction to the automaton in Figure 2.4. We begin with the starting state $\{q_0\}$.



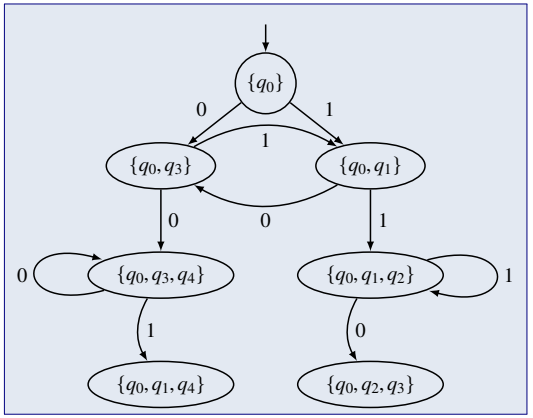
Since $\delta(q_0, 0) = \{q_0, q_3\}$ and $\delta(q_0, 1) = \{q_0, q_1\}$, we extend the graph as follows:



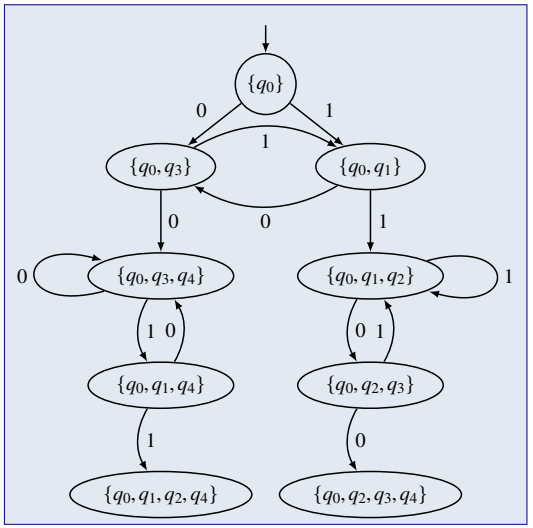
We extend the graph according to the outgoing transitions from the states inside the newly generated nodes $\{q_0, q_3\}$ and $\{q_0, q_1\}$:



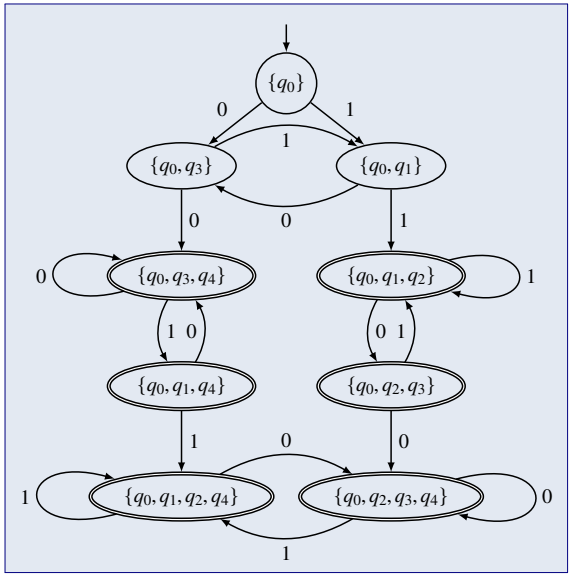
We extend the graph according to the outgoing transitions from the states inside the newly generated nodes $\{q_0, q_3, q_4\}$ and $\{q_0, q_1, q_2\}$:



We extend the graph by transitions from the new nodes:



We extend again by transitions from the new nodes; from these nodes, no new transitions can be generated, so we finally mark the accepting nodes:



The result is a deterministic automaton whose first three states correspond to those of the previously given one; the remaining six states are all accepting states that cannot be left by any transition; these correspond to the two accepting states of the previous automaton that could also not be left. We see that, while the subset construction leads to a deterministic automaton, the result is not necessarily optimal. □

2.3. Minimization of Finite State Machines

As could be seen from the example in the previous section, multiple deterministic automata can accept the same language, but considerably differ in their sizes (number of states). In this section, we will discuss how to minimize the number of states in an automaton while preserving the language that it accepts.

The construction depends on a notion of "equivalence" of states:

Definition 10 (State Equivalence, Bisimulation) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFSM. We define the relation $\sim_k$ of M, a binary relation on state set Q, by induction on k:

$$q_1 \sim_0 q_2 :\Leftrightarrow (q_1 \in F \Leftrightarrow q_2 \in F)$$

$$q_1 \sim_{k+1} q_2 :\Leftrightarrow \forall a \in \Sigma : \delta(q_1, a) \sim_k \delta(q_2, a)$$

We define the the *bisimulation* relation $\sim$ of M, a binary relation on Q, as

Bisimulation

$$q_1 \sim q_2 \Leftrightarrow \forall k \in \mathbb{N} : q_1 \sim_k q_2$$

If $q_1 \sim q_2$, we say that states q_1 and q_2 are *state equivalent*.

(zustands)äquivalent

By this definition we have $q_1 \sim_k q_2$ if starting with q_1 the same words of length k are accepted as if starting with q_2. We have $q_1 \sim q_2$ if the same words of arbitrary length are accepted. It can be shown that, if $q_1 \sim_k q_2$ for $k = |Q|$, then $q_1 \sim q_2$, i.e., if two states are equivalent for words whose length equals the number of states, then the two states are equivalent for words of arbitrary length.

If an automaton has two equivalent states, they may be merged into one; the resulting automaton is smaller but has the same language as the original one. Based on this principle, we will construct an algorithm for the minimization of automata. This algorithm decomposes the state set of the given automaton into partitions (clusters) of equivalent states. Each partition then represents a state of the minimized automaton.

We start with an auxiliary notion which is illustrated in Figure 2.5.

Definition 11 (State Partition) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFSM. Let $S \subseteq P(Q)$ be a set of partitions of state set Q, $p \in S$ be a partition in S, and $s \in p$ be a state in p. Then the *state partition* $[s]_p^S$ is defined as

(Zustands)partition

$$[s]_p^S = \{\, t \in p \mid \forall a \in \Sigma, q \in S : \delta(t, a) \in q \Leftrightarrow \delta(s, a) \in q \}$$

In other words, $[s]_p^S$ consists of all those states of p from which every transition leads to the same partition in S as the corresponding transition does from state s.

Let us assume we have a set S of partitions such that the elements in each partition are in relation $\sim_k$ (for some $k \in \mathbb{N}$), i.e., they cannot be distinguished by any word of length k. Given a partition $p \in S$ and a state $s \in p$, we can by the notion introduced above split p into two partitions: one is $[s]_p^S$ while the other one contains all other states of p. Since all states in the

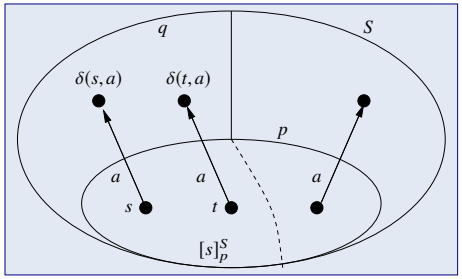


Figure 2.5.: A State Partition

original partition p are in relation $\sim_k$, i.e., they cannot be distinguished from s by any word of length k, the states of $[s]_p^S$ are in relation $\sim_{k+1}$. This is because the application of the transition function reaches for all input symbols states which are in the same target partition and therefore cannot be distinguished by any input word of length k.

On the other hand, the states in $[s]_p^S$ can be distinguished from the other states in p by a word of length $k+1$, since for some input symbol the transition function yields two states which are in different partitions and therefore can be distinguished by input words of length k. If we construct $[s]_p^S$ for *every* state $s \in p$, we can thus "break up" p into a number of partitions such that the states in each partition are exactly those states which are in relation $\sim_{k+1}$.

Based on this idea, the algorithm Partition in Figure 2.6 partitions the state set of a DFSM into clusters of equivalent states by repeatedly breaking up previously constructed clusters of states which are in relation $\sim_k$ into clusters of states in relation $\sim_{k+1}$ until the states in the clusters are in relation $\sim$, i.e., they are equivalent. In more detail, starting with the partitioning $P_0 = \{F, Q\backslash F\}$ (whose elements consist of those states that are in relation $\sim_0$), the algorithm constructs a sequence of partitionings $P_0, P_1, P_2, \ldots, P_n$ of state set Q where every P_k consists of those clusters whose elements are in relation $\sim_k$. We have $n \leq |Q|$, because $\sim_{|Q|}$ equals $\sim$, i.e., after at most $|Q|$ iterations no new cluster can be constructed and the algorithm terminates.

Using Partition as a sub-algorithm, the algorithm Minimize in Figure 2.6 minimizes a given DFSM. First, Minimize removes all unreachable states from state set Q of the original automaton M and partitions the remaining states into clusters of equivalent states. These clusters become the states of the minimized automaton M'; the initial state q_0' is the cluster which contains the initial state q_0 of M; the set of accepting states F' contains those clusters that hold some accepting state of M. The transition relation δ' of M' maps cluster q' and symbol a to that cluster to which all the elements q of q' are mapped by the transition relation M of a. Such a cluster exists because the elements of every cluster are state equivalent. The result is an


```
function Partition(Q, Σ, δ, q0, F)
    P ← {F, Q\F}
    repeat
        S ← P
        P ← ∅
        for p ∈ S do
            P ← P ∪ {[s]_p^S | s ∈ p}
        end for
    until P = S
    return P
end function
```

```
function Minimize(Q, Σ, δ, q0, F)
    Q ← {q ∈ Q | ∃w ∈ Σ* : δ*(q0, w) = q}
    Q' ← Partition(Q, Σ, δ, q0, F)
    for q' ∈ Q', a ∈ Σ do
        set δ'(q', a) to that partition q'' of Q'
            where ∀q ∈ q' : δ(q, a) ∈ q''
    end for
    let q0' be that partition of Q' where q0 ∈ q0'
    F' ← {q ∈ Q' : q ∩ F ≠ ∅}
    return (Q', Σ, δ', q0', F')
end function
```

Figure 2.6.: Minimization of a DFSM

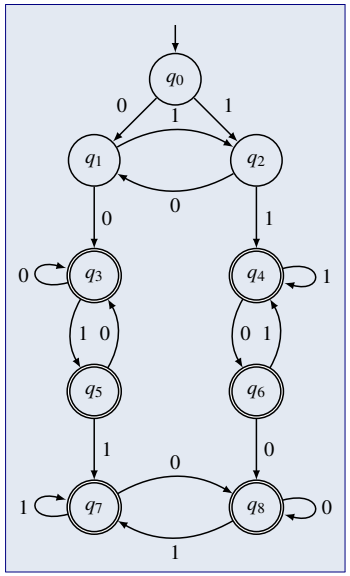
automaton which has different states q_1 and q_2 only if $q_1 \not\sim q_2$.

Theorem 2 (DFSM *Partitioning* and *Minimization*) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFSM. Zerlegung Minimierung

1. Partition(M) returns the smallest partitioning of the state set of M into clusters of equivalent states.
2. Minimize(M) returns a DFSM with the smallest number of states whose language is $L(M)$.

Based on above concepts, we can also determine whether two automata M_1 and M_2 have the same language: we apply algorithm Partition to a combination of M_1 and M_2, i.e., an automaton whose state set is the (disjoint) union of the state sets of M_1 and M_2: if the start states q_0 of M_1 and q_0' of M_2 end up in the same cluster, we have $q_0 \sim q_0$ and thus $L(M_1) = L(M_2)$.

Example 6 (Minimization) Take the DFSM constructed in Example 5 from a NFSM by the subset construction (we have renamed the states for simplicity):



The minimization algorithm starts with a partitioning into two clusters

$$p_0 := \{q_0, q_1, q_2\}$$
$$p_3 := \{q_3, q_4, q_5, q_6, q_7, q_8\}$$

All transitions from a state in p_3 lead to cluster p_3, thus this cluster need not be split any more.

However we have to further split p_0: since

$$\delta(q_0, 0) = q_1 \in p_0$$
$$\delta(q_1, 0) = q_3 \in p_3$$

we have to put q_0 and q_1 into different clusters. Likewise, since

$$\delta(q_0, 1) = q_2 \in p_0$$
$$\delta(q_2, 1) = q_4 \in p_3$$

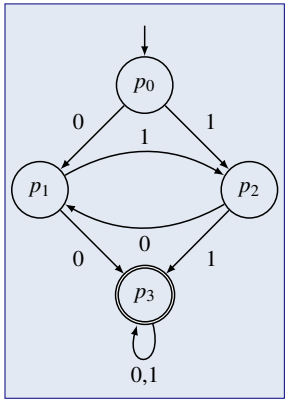
we have to put q_0 and q_2 into different clusters. Finally, since

$$\delta(q_1, 0) = q_3 \in p_3$$
$$\delta(q_2, 0) = q_1 \in p_0$$

we have to put q_1 and q_2 into different clusters. Consequently, cluster p_0 is split into three singleton clusters:

$$p_0 := \{q_0\}$$
$$p_1 := \{q_1\}$$
$$p_2 := \{q_2\}$$

These clusters cannot be split any more, thus the algorithm terminates. The minimized automaton is as follows:



This automaton is even smaller than the DFSM presented in Example 4 which accepts the same language of words of 0 and 1 which contain 00 or 11. □

2.4. Regular Languages and Finite State Machines

We are now going to introduce a mechanism to define the class of *regular languages*.

Definition 12 (Regular Expression) Let Σ be an alphabet. The set of *regular expressions* $Reg(\Sigma)$ over Σ is inductively defined as follows:

regulärer Ausdruck

- $\emptyset \in Reg(\Sigma)$ and $\varepsilon \in Reg(\Sigma)$.
- If $a \in \Sigma$, then $a \in Reg(\Sigma)$.
- If $r_1, r_2 \in Reg(\Sigma)$, then $(r_1 \cdot r_2) \in Reg(\Sigma)$ and $(r_1 + r_2) \in Reg(\Sigma)$.
- If $r \in Reg(\Sigma)$, then $(r^*) \in Reg(\Sigma)$.

For $\Sigma = \{a_1, \ldots, a_n\}$, the set $Reg(\Sigma)$ can be also described by the following grammar rule

$$r ::= \emptyset \mid \varepsilon \mid a_1 \mid \ldots \mid a_n \mid (r \cdot r) \mid (r + r) \mid (r^*)$$

To avoid the heavy use of parenthesis, we assume that * binds stronger than $\cdot$ which in turn binds stronger than +; e.g. rather than writing

$$(a + (b \cdot (c^*)))$$

we typically write

$$a + b \cdot c^*$$

We also often drop the $\cdot$ symbol and just write

$$a + bc^*$$

Definition 13 (Language Concatenation and Closure) Let Σ be an alphabet and $L_1, L_2 \subseteq \Sigma^*$ be languages over Σ. We define the following operations on languages:

(Sprach)verkettung
Verkettung

- The *language concatenation* (short *concatenation*)

$$L_1 \circ L_2 := \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$$

endlicher (Sprach)abschluss
endlicher Abschluss

- The *finite language closure* (short *finite closure*)

$$L^* := \bigcup_{i=0}^{\infty} L^i$$

(i.e., $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$) where

$$L^0 := \{\varepsilon\}$$
$$L^{i+1} := L \circ L^i$$

Thus $L_1 \circ L_2$ contains all words that can be decomposed in a prefix from L_1 and a suffix from L_2; furthermore, L^* contains all words that are concatenations of finitely many words from L (including the empty word ε which represents the zero-fold concatenation).

Definition 14 (Language of Regular Expressions) Let Σ be an alphabet. Then the *regular expression language* $L(r) \subseteq \Sigma^*$ of regular expression $r \in Reg(\Sigma)$ is inductively defined as follows:

die Sprache eines regulären Ausdrucks

- $L(\emptyset) := \emptyset$.
- $L(\varepsilon) := \{\varepsilon\}$.
- $L(a) := \{a\}$.
- $L(r_1 \cdot r_2) := L(r_1) \circ L(r_2)$.
- $L(r_1 + r_2) := L(r_1) \cup L(r_2)$.
- $L(r^*) := L(r)^*$.

We notice $L(r_1 \cdot (r_2 \cdot r_3)) = L((r_1 \cdot r_2) \cdot r_3)$, i.e., the concatenation operator "$\cdot$" is associative with respect to language building. Therefore the setting of parentheses in a concatenation does not matter and we typically write $r_1 \cdot r_2 \cdot r_3$ or just $r_1 r_2 r_3$.

Definition 15 (Regular Language) Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language over Σ. Then L is a *regular language* (over Σ), if there exists a regular expression $r \in Reg(\Sigma)$ over Σ such that

reguläre Sprache

$$L = L(r)$$

Example 7 (Regular Languages) The language of the automaton depicted in Figure 2.2 is regular; it is denoted by the regular expression

$$(a + b)(a + b + 0 + 1)^*$$

Also the language of the automaton depicted in Figure 2.4 is regular; it is denoted by the regular expression

$$(0 + 1)^*(00 + 11)(0 + 1)^*$$

The language of the vending machine depicted at the beginning of this chapter is regular; it is denoted by the regular expression

(50¢ abort)*(1€ + 50¢ 50¢) □

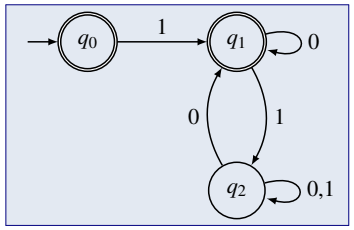
Above example shows that the languages of some automata are regular; one might wonder whether this is the case for all automata (which is not self-evident, e.g., it is hard to see what the regular expression for the language of the automaton depicted in Figure 2.3 might be).

Also the converse question might be of interest: given a regular expression, does there exist an automaton recognizing this language?

Example 8 (Regular Languages) Let r be the regular expression

$$\varepsilon + 1(0 + 1(0 + 1)^*0)^*$$

Then the following NFSM M has language $L(r)$.



Since $\varepsilon \in L(r)$, the start state q_0 of M is accepting. Since $1 \in L(r)$, we have a transition from q_0 to another accepting state q_1. Since the 1 may be followed by an arbitrary repetition of 0, we have a transition from q_1 to itself. Since any element of this repetition may be also 10, we have another path from q_1 to itself via intermediate state q_2. Since the two symbols of 10 may be separated by arbitrary many repetitions of 0 or 1, we have corresponding transitions from q_2 to itself. □

Again, the question arises whether above example demonstrates a general principle, i.e., whether for every regular expression an automaton exists that has the same language as the expression. Actually, for this question as well as for the one raised above, the answer is "yes".

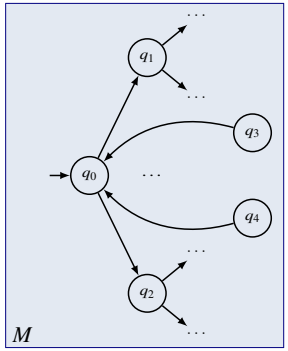
Theorem 3 (Equivalence of Regular Expressions and Automata)

1. For every regular expression r over Σ, there exists an automaton M with input alphabet Σ such that $L(M) = L(r)$.
2. For every automaton M with input alphabet Σ, there exists a regular expression r over Σ such that $L(r) = L(M)$.

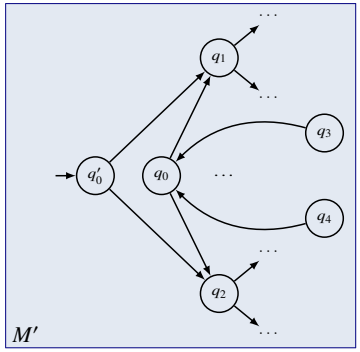
We will sketch the proof of this theorem by the construction of an automaton from a regular expression and vice versa.

Proof (Proof of Part 1 of Theorem 3) Let r be a regular expression over Σ; we show by induction on the structure of r the construction of a NFSM M with input alphabet Σ such that $L(M) = L(r)$. M has exactly one start state and arbitrarily many accepting states (one of which may be also the start state).

In the course of the following constructions, it is sometimes necessary to require from an automaton M that it does not have transitions back to its the start state. If M happens to have such transitions, i.e., if it has form



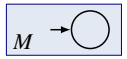
then we can replace M by the following automaton M' that has the same language as M but not any transitions back to its start state:



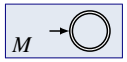
Essentially, M' adds a new start state q_0' that is accepting, if and only if q_0 is accepting, and that has transitions to the same states as the original start state q_0; all of the transitions back to q_0 are thus not any more transitions to the start state.

The construction of M from r is performed according to the following cases:

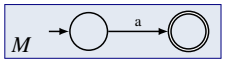
- **Case $\mathbf{r} = \emptyset$:** M is the following automaton with a single non-accepting state:



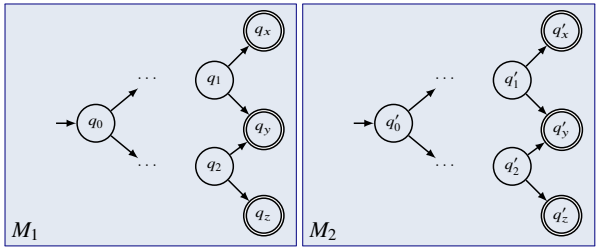
- **Case $\mathbf{r} = \varepsilon$:** M is the following automaton with a single accepting state:



- **Case $\mathbf{r} = \mathbf{a}$ (for some $a \in \Sigma$):** M is the following automaton with two states, one of which is accepting:



- **Case $\mathbf{r} = \mathbf{r_1} \cdot \mathbf{r_2}$:** By the induction hypothesis, we have two automata M_1 and M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The automata have general form



  where q_0 is the start state of M_1 and q'_0 is the start state of M_2 (both may or may not be accepting). We may assume that there are no transitions to the start states q'_0 of M_2; otherwise M_2 has to be transformed as explained above.

  Here q_1, q_2 in M_1 represent all the nodes (there may be arbitrarily many) from which there is a transition to some accepting state q_x, q_y, q_z in M_1 (there may be arbitrarily many). Likewise, q'_1, q'_2 in M_2 represent all the nodes (there may be arbitrarily many) from which there is a transition to some accepting state q'_x, q'_y, q'_z of M_2.

  As shown in Figure 2.7, automaton M is then defined by linking a copy of M_2 to every accepting state q_x, q_y, q_z of M_1, i.e., in every copy of M_2, q'_0 is identified with one of these states; q_x, q_y, q_z become accepting in M if q'_0 is accepting in M_2.

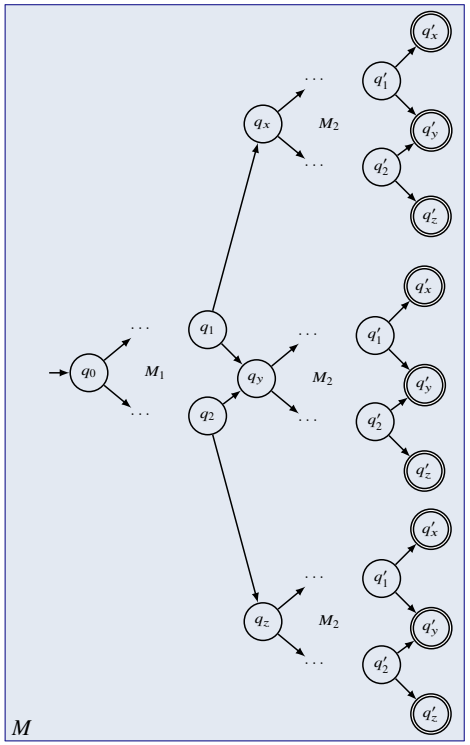


Figure 2.7.: Case $r = r_1 \cdot r_2$

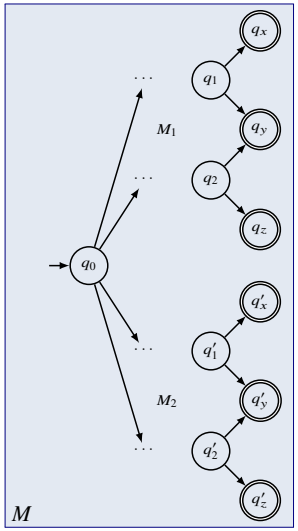
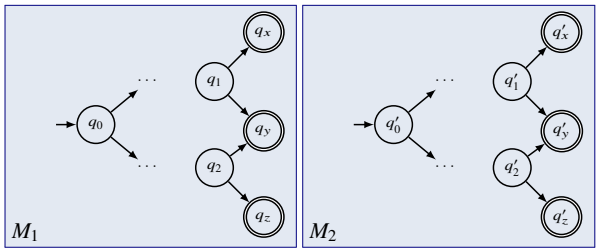


Figure 2.8.: Case $r = r_1 + r_2$

Due to this linkage, it is important that M_2 does not have a transition back to its start state: otherwise M might, after following a path through M_2, return to some of the states q_x, q_y, q_z and then (since there may exist in M_1 paths that lead from these states back to themselves) continue by following a path through M_1 before returning to M_2. Consequently, executions of M_1 and M_2 might be arbitrarily interleaved such that $L(M) \neq L(M_1) \circ L(M_2)$. Our construction therefore ensures that after M has entered the execution of some copy of M_2 no more execution of some part of M_1 is possible.

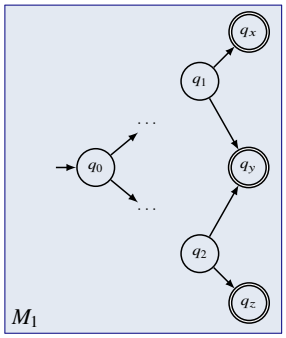
- **Case $\mathbf{r = r_1 + r_2}$:** By the induction hypothesis, we have two automata M_1 and M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The automata have general form



where we may assume that there are no transitions to the start states q_0 of M_1 and q'_0 of M_2 (which may or may not be accepting); otherwise M_1 respectively M_2 have to be transformed as explained above.

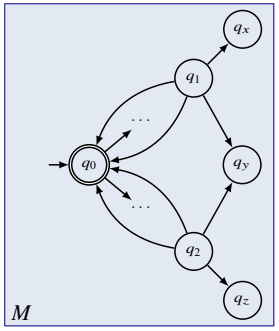
As shown in Figure 2.8, automaton M is then constructed by parallel composition of M_1 and M_2, i.e., their start states q_0 and q'_0 are identified; this state becomes accepting in M if q_0 in M_1 or q'_0 in M_2 is accepting. Due to this composition, it is important that M_1 and M_2 must not return to their start states because otherwise executions of M_1 and M_2 might be arbitrarily interleaved.

- **Case $\mathbf{r = r_1^*}$:** By the induction hypothesis, we have an automaton M_1 such that $L(M_1) = L(r_1)$. This automaton has general form

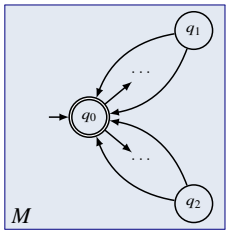


where the start state q_0 of M_1 may or may not be accepting.

Then M has the following form where for every transition to an accepting state in M_1 an additional transition to q_0 is created; furthermore, q_0 becomes the only accepting state of M:



Now those nodes q_x, q_y, q_z (which were accepting in M_1 but are not any more in M) can be erased from which no path leads to q_0 ; if this is the case for all of them, the resulting automaton has the following form:

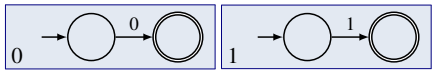


This construction is the only one that introduces transitions to the start state; if the resulting automaton shall serve as a building block for the composition of another automaton from a regular expression that contains r_1^*, then a new start state q_0' has to be introduced as described in the beginning of the proof.

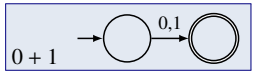
While the correctness of the constructions seems pretty self-evident, it is still necessary to verify that the languages of the constructed automata represent the languages of the corresponding regular expressions (we omit the details). □

Example 9 (Automaton from Regular Expression) We apply above construction to the regular expression $(0 + 1)^*(00 + 11)(0 + 1)^*$.

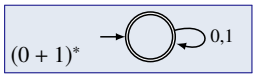
The automata corresponding to the regular expressions 0 and 1 are



The automaton for $0 + 1$ then is

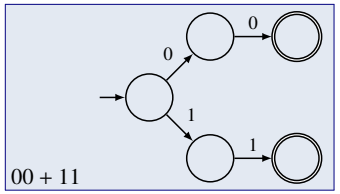


From this, we construct the automaton for $(0 + 1)^*$

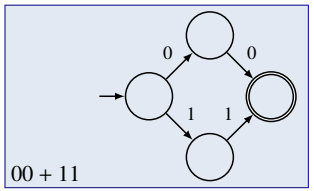


The automata for 00 and 11 are

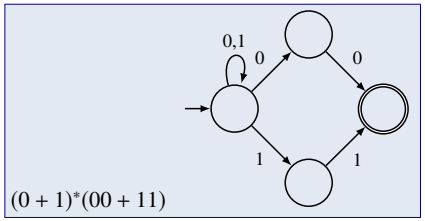
From these, we can construct the automaton for $00 + 11$ as



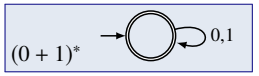
which can be simplified to



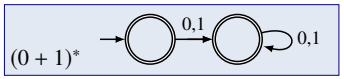
Now we construct the automaton for $(0 + 1)^*(00 + 11)$ as



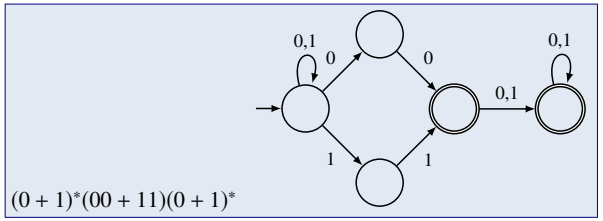
Since the construction of the automaton for $(0 + 1)^*(00 + 11)(0 + 1)^*$ from the automata for $(0 + 1)^*(00 + 11)$ and $(0 + 1)^*$ demands that the automaton for $(0 + 1)^*$ does not have a transition to its start state, we transform the automaton



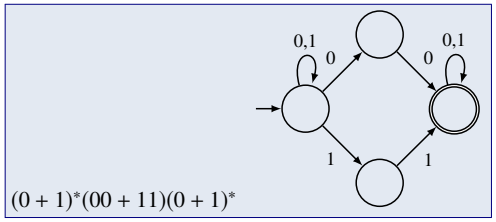
constructed above accordingly:



The automaton for $(0+1)^*(00+11)(0+1)^*$ can now be constructed as



which can be further simplified to



The result is an optimized version of the automaton presented in Figure 2.4. □

Using the constructions presented up to now, a regular expression can be transformed to a minimized deterministic automaton by the following sequence of steps:

regular expression $\overset{\text{Theorem 3}}{\longrightarrow}$ NFSM $\overset{\text{Theorem 1}}{\longrightarrow}$ DFSM $\overset{\text{Theorem 2}}{\longrightarrow}$ minimized DFSM

We are now going to complete the proof of Theorem 3.

Proof (Proof of Part 2 of Theorem 3) Let $M = (Q, \sigma, \delta, q_0, F)$ be a DFSM; we show the construction of a regular expression r over Σ such that $L(r) = L(M)$.

We start by defining $R_{q,p}$ as the set of all words that drive M from state q to state p:

$$R_{q,p} := \{w \in \Sigma^* \mid \delta^*(q,w) = p\}$$

Assume that, for arbitrary states $q, p \in Q$, we can construct a regular expression $r_{q,p}$ with $L(r_{q,p}) = R_{q,p}$. We can then define the regular expression r as

$$r := r_{q_0,p_1} + \ldots + r_{q_0,p_n}$$

where $p_1, \ldots, p_n$ are all states of F. The language of M is the set of all words that drive M from start state q_0 to some end state p, i.e.

$$L(M) = R_{q_0,p_1} \cup \ldots \cup R_{q_0,p_n}$$

Thus by the construction of r and the definition of $L(r)$, we know $L(r) = L(M)$.

We now show that such $r_{q,p}$ can indeed be constructed. Let $a_1, \ldots, a_n \in \Sigma$ denote those symbols (in some permutation) that drive M from q to p, i.e., $\delta(q, a_i) = p$ for $1 \le i \le n$. Let $q_0, \ldots, q_{|Q|-1}$ denote the elements of Q (in some permutation) and let,, for $0 \le j \le |Q|$, Q_j denote the subset $\{q_0, \ldots, q_{j-1}\}$ of Q. Then we define, for $0 \le j \le |Q|$, the set $R^j_{q,p}$ as:

$$R^0_{q,p} := \begin{cases} \{a_1, \ldots, a_n\}, & \text{if } q \neq p \\ \{a_1, \ldots, a_n, \varepsilon\}, & \text{if } q = p \end{cases}$$
$$R^{j+1}_{q,p} := \{w \in R_{q,p} \mid \forall 1 \le k \le |w| : \delta^*(q, w \downarrow k) \in Q_{j+1}\}$$

Thus $R^0_{q,p}$ consists of those words of length zero or one that drive M from q to p. For $j > 0$, set $R^j_{q,p}$ consists of those words that drive M from q to p such that M passes only states in Q_j.

Assume that, for arbitrary states $q, p \in Q$ and $0 \le j \le |Q|$, we can construct a regular expression $r^j_{q,p}$ with $L(r^j_{q,p}) = R^j_{q,p}$. We can then define the regular expression $r_{q,p}$ as

$$r_{q,p} := r^{|Q|}_{q,p}$$

because $|Q| > 0$ and, by the definition of $R^{j+1}_{q,p}$, thus $R_{q,p} = R^{|Q|}_{q,p}$.

We now define $r^j_{q,p}$ by induction on j:

$$r^0_{q,p} := \begin{cases} \emptyset, & \text{if } q \neq p \wedge n = 0 \\ a_1 + \ldots + a_n, & \text{if } q \neq p \wedge n \ge 1 \\ a_1 + \ldots + a_n + \varepsilon, & \text{if } q = p \end{cases}$$
$$r^{j+1}_{q,p} := r^j_{q,p} + r^j_{q,q_j} \cdot (r^j_{q_j,q_j})^* \cdot r^j_{q_j,p}$$

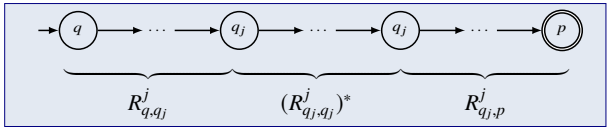
From the definition of $r^0_{q,p}$ and $R^0_{q,p}$, we have $L(r^0_{q,p}) = R^0_{q,p}$.

It remains to show that $L(r^{j+1}_{q,p}) = R^{j+1}_{q,p}$. By the definition of $r^{j+1}_{q,p}$, it suffices to show

$$R^j_{q,p} \cup R^j_{q,q_j} \circ (R^j_{q_j,q_j})^* \circ R^j_{q_j,p} = R^{j+1}_{q,p}$$

We thus have to show that a word is in set $R^j_{q,p} \cup R^j_{q,q_j} \circ (R^j_{q_j,q_j})^* \circ R^j_{q_j,p}$ if and only if it is in

set $R^{j+1}_{q,p}$. This is true, because, if a word drives M from state p to state q via states in Q_{j+1}, it either drives M from p to q only via states in Q_j or we have at least one occurrence of state $q_j \in Q_{j+1} \backslash Q_j$ along the path:



In that case, the word consists of a prefix that drives M from q to the first occurrence of q_j only via states in Q_j, a middle part that drives M repeatedly from one occurrence of q_j to the next occurrence of q_j only via states in Q_j and a suffix that drives M from the last occurrence of q_j to p only via states in Q_j. □

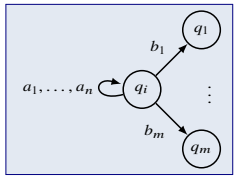
Above construction of a regular language from an automaton is quite tedious, but there is a much simpler way based on the following result which was proved by D.N. Arden in 1960.

Theorem 4 (Arden's Lemma) Let L, U, V be regular languages with $\varepsilon \notin U$. Then

$$L = U \circ L \cup V \Leftrightarrow L = U^* \circ V$$

The relevance of this lemma (which we state without proof) is that it gives us the solution of a recurrence $L = U \circ L \cup V$ for the unknown L as $L = U^* \circ V$; in an analogous way we can compute the solution of a regular expression recurrence $l = u \cdot l + v$ (i.e., a regular expression l which satisfies the property $L(l) = L(u \cdot l + v)$) as $l = u^* \cdot v$.

We can apply this solution technique to the construction of regular expressions from automata in the following way: for every state q with outgoing transitions to q itself via symbols $a_0, \ldots, a_n$ and to other states $q_1, \ldots, q_n$ via symbols $b_1, \ldots, b_m$



we construct an equation

$$X_i = (a_1 + \ldots + a_n) \cdot X_i + b_1 \cdot X_1 + \ldots + b_m \cdot X_m$$

Here X_i represents the regular expression of the language that the automaton accepts starting with state X_i and the other variables represent the analogous regular expressions for their respective states. As an exception, for every accepting state q_a, the equation has form

$$X_a = (a_1 + \ldots + a_n) \cdot X_a + b_1 \cdot X_1 + \ldots + b_m \cdot X_m + \varepsilon$$

because from state q_a also the empty word ε is accepted.

In the same way, we can construct an equation for every state of the automaton; if the automaton has s states, we thus get a system of s equations in s unknowns. Our goal is to solve this equation system for X_0, i.e., to compute the regular expression for the initial state q_0. By Arden's Lemma, we can compute for any X_i the solution

$$X_i = (a_1 + \ldots + a_n)^* \cdot (b_1 \cdot X_1 + \ldots + b_m \cdot X_m)$$

respectively for a variable X_a corresponding to an accepting state

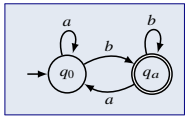
$$X_a = (a_1 + \ldots + a_n)^* \cdot (b_1 \cdot X_1 + \ldots + b_m \cdot X_m + \varepsilon)$$

We can substitute this solution in all the other equations and thus get a system of $s - 1$ equations in $s - 1$ unknowns. After the substitution, we choose the equation for some other variable X_j. In this equation, we apply the regular expression transformations

$$a \cdot \varepsilon = a$$
$$\varepsilon \cdot a = a$$
$$a \cdot (b + c) = a \cdot b + a \cdot c$$
$$(a + b) \cdot c = a \cdot c + b \cdot c$$

(these transformations preserve the languages of the regular expressions as can be easily shown from their definitions) to yield again an equation of the form to which Arden's Lemma can be applied. We repeat the process until we have only one equation in the single unknown X_0; the solution of this equation is the regular expression of the language accepted by the automaton.

Example 10 Consider the automaton M



From M, we construct the equation system

$$X_0 = a \cdot X_0 + b \cdot X_a$$
$$X_a = b \cdot X_a + a \cdot X_0 + \varepsilon$$

By Arden's Lemma we solve the second equation as

$$X_a = b^* \cdot (a \cdot X_0 + \varepsilon)$$

By substituting this solution into the first equation and further transformation, we get

$$\begin{aligned} X_0 &= a \cdot X_0 + b \cdot b^* \cdot (a \cdot X_0 + \varepsilon) \\ &= a \cdot X_0 + b \cdot b^* \cdot a \cdot X_0 + b \cdot b^* \cdot \varepsilon \\ &= (a + b \cdot b^* \cdot a) \cdot X_0 + b \cdot b^* \end{aligned}$$

By applying Arden's Lemma again, we get

$$X_0 = (a + b \cdot b^* \cdot a)^* \cdot b \cdot b^*$$

such that $L(X_0) = L(M)$. □

A consequence of the translation of automata to regular expressions is the insight that, while automata are powerful enough to recognize every regular language, they are not powerful enough to recognize any other language. Languages which cannot be described by regular expressions are therefore also not in the scope of automata. We will therefore investigate the expressiveness of regular languages further.

2.5. The Expressiveness of Regular Languages

We start with a result that shows how new regular languages can be constructed from existing ones, i.e., that the domain of regular languages is closed under certain building operations.

Abschlusseigenschaften

Theorem 5 (*Closure properties* of Regular Languages) Let Σ be an alphabet and L, L_1, L_2 be regular languages over Σ. Then the following languages are also regular languages over Σ:

1. the complement $\overline{L} = \{x \in \Sigma^* \mid x \notin L\}$;
2. the union $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$;
3. the intersection $L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$;
4. the concatenation $L_1 \circ L_2$;
5. the finite language closure L^*.

Proof Let Σ be an alphabet and L, L_1, L_2 be regular languages over Σ.

1. Since L is a regular language over Σ, because of Part 1 of Theorem 3, there exists a DFSM $M = (Q, \Sigma, \delta, q_0, F)$ and $L = L(M)$. We can define a DFSM M' which is identical to M, except that the set of accepting states F' of M' is the complement of F, i.e., $F' = Q \backslash F$. Thus M' accepts every word over Σ^* that is *not* accepted by M, i.e. $L(M') = \overline{L}$. Because of Part 2 of Theorem 3, thus $\overline{L}$ is a regular language over Σ.
2. Since L_1 and L_2 are regular languages over Σ, there exist regular expressions r_1 and r_2 over Σ, such that $L_1 = L(r_1)$ and $L_2 = L(r_2)$. Then $r_1 + r_2$ is a regular expression over Σ with $L(r_1 + r_2) = L(r_1) \cup L(r_2) = L_1 \cup L_2$, thus $L_1 \cup L_2$ is a regular language over Σ.
3. We know $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. By parts 1 and 2 of Theorem 5, thus $L_1 \cap L_2$ is a regular language over Σ.
4. Since L_1 and L_2 are regular languages over Σ, there exist regular expressions r_1 and r_2 over Σ such that $L_1 = L(r_1)$ and $L_2 = L(r_2)$. Then $r_1 \cdot r_2$ is a regular expression over Σ with $L(r_1 \cdot r_2) = L(r_1) \circ L(r_2) = L_1 \circ L_2$, thus $L_1 \circ L_2$ is a regular language over Σ.
5. Since L is a regular language over Σ, there exists a regular expressions r over Σ such that $L = L(r)$. Then r^* is a regular expression over Σ with $L(r^*) = L(r)^* = L^*$, thus L^* is a regular language over Σ. □

While regular languages can thus be constructed in a quite flexible way, their expressiveness is limited by the following result.

Pumping-Lemma (Schleifensatz)

Aufpumplänge

Theorem 6 (*Pumping Lemma*) Let L be a regular language. Then there exists a natural number n (called the *pumping length* of L) such that every word $w \in L$ with $|w| \geq n$ can be decomposed into three substrings x, y, z, i.e. $w = xyz$, such that

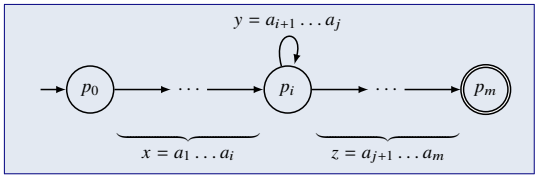
1. $|y| \geq 1$,
2. $|xy| \leq n$,
3. $xy^k z \in L$, for every $k \geq 0$.

Thus every sufficiently large word of a regular language has a middle part that can be repeated arbitrarily often ("pumped") to yield another word of the language.

Proof Let L be a regular language and $M = (Q, \Sigma, \delta, q_0, F)$ a DFSM with $L = L(M)$. We define $n := |Q|$ as the number of states in M. We take arbitrary $w \in L$ with $|w| \geq n$, i.e. $w = a_1 a_2 \dots a_m$ with $m \geq n$ and define, for every $0 \leq i \leq m$, p_i as

$$p_0 := q_0$$
$$p_{i+1} := \delta(p_i, a_{i+1})$$

i.e. $p_0, p_1, \dots, p_m$ is the sequence of states that M passes when accepting w. Since $m \geq n$, there must exist two positions i and j with $0 \leq i < j \leq m$ such that $p_i = p_j$, i.e., the sequence contains a cycle from p_i to p_i:



Now define $x := a_1 \dots a_i$, $y := a_{i+1} \dots a_j$, $z := a_{j+1} \dots a_m$, i.e., x is the prefix of w read when M runs from p_0 to p_i, y is the middle part of w read when M runs from p_i to p_i, and z is the suffix of w read when M runs from p_i to the accepting state. Since the state sequence $p_i, \dots, p_i$ forms a cycle, M accepts, for every $k \geq 0$, also the word $xy^k z$. Thus $xy^k z \in L(M) = L$. □

The Pumping Lemma can be used to show that a given language L is *not* regular. The usual technique is to assume that L is regular, take a sufficiently long word in L, and show that, by application of the lemma, from this word a longer word can be constructed whose form contradicts the definition of L; this invalidates the assumption that L is regular.

Example 11 (Pumping Lemma) Let $L = \{0^i 1^i \mid i \in \mathbb{N}\}$, i.e. L consists of all words that can be split into a prefix which contains only 0 and a suffix which contains only 1 such that both parts have the same length:

$$L = \{\varepsilon, 01, 0011, 000111, \dots\}$$

We show that L is not regular. Assume that L is regular. Let n be the pumping length of L. Take the word $w := 0^n 1^n$. Since w is by construction in L and satisfies the requirement $|w| \geq n$, we can split w into three parts x, y, z that have the following properties:

$$xyz = 0^n 1^n \tag{1}$$
$$|y| \geq 1 \tag{2}$$
$$|xy| \leq n \tag{3}$$

Properties (1) and (3) imply that x and y contain 0 only, while z can contain 0 and 1. Thus we have four values $n_1, n_2, n_3, n_4 \in \mathbb{N}$ such that $x = 0^{n_1}$, $y = 0^{n_2}$, $z = 0^{n_3} 1^{n_4}$. From the definition of L and Property (2), we know

$$n_1 + n_2 + n_3 = n_4 \tag{4}$$
$$n_2 \geq 1 \tag{5}$$

By the Pumping Lemma, we know $xy^2 z \in L$, i.e,

$$0^{n_1} 0^{2n_2} 0^{n_3} 1^{n_4} \in L \tag{6}$$

This implies, by the definition of L,

$$n_1 + 2n_2 + n_3 = n_4 \tag{7}$$

But we also know

$$n_1 + 2n_2 + n_3 = (n_1 + n_2 + n_3) + n_2 \overset{(4)}{=} n_4 + n_2 \overset{(5)}{\geq} n_4 + 1 > n_4 \tag{8}$$

Properties (7) and (8) represent a contradiction. □

Example 12 (Pumping Lemma) Let $L = \{0^{i^2} \mid i \in \mathbb{N}\}$, i.e., L consists of all words that

contain only 0 and whose length is a square number:

$$L = \{\varepsilon, 0, 0000, 000000000, \ldots\}$$

We show that L is not regular. Assume that L is regular. Let n be the pumping length of L.

Take the word $w := 0^{n^2}$. Since w is by construction in L and satisfies the requirement $|w| \geq n$, we can split w into three parts x, y, z that have the following properties:

$$xyz = 0^{n^2} \tag{1}$$
$$|y| \geq 1 \tag{2}$$
$$|xy| \leq n \tag{3}$$

where (3) implies

$$|y| \leq n \tag{4}$$

Then the Pumping Lemma implies that also the word $xy^2z \in L$. Since $|xy^2z| = |xyz| + |y| = n^2 + |y|$, this implies that $n^2 + |y|$ is a square number. But we know

$$n^2 < n^2 + 1 \overset{(2)}{\leq} n^2 + |y| \tag{5}$$
$$n^2 + |y| \overset{(4)}{\leq} n^2 + n < n^2 + 2n + 1 = (n+1)^2 \tag{6}$$

Together (5) and (6) imply

$$n^2 < n^2 + |y| < (n+1)^2 \tag{7}$$

Thus $n^2 + |y|$ is not a square number, which represents a contradiction. □

These examples show that the expressive power of regular languages is limited. They cannot describe languages whose construction principles depend on more complex properties (such as general arithmetic) as can be formulated by regular expressions[1]. Such languages can therefore also not be recognized by finite-state machines. In the following, we will therefore turn our attention to more powerful computational models.

[1] However, be aware that *some* arithmetic properties *can* be used to construct regular languages. For instance, the language $\{0^i \mid i \text{ is even}\} = \{\varepsilon, 00, 0000, 000000, \ldots\}$ is regular, because it is the language of the regular expression $(00)^*$.

Chapter 3.

Turing Complete Computational Models and Recursively Enumerable Languages

As shown in Chapter 2, the expressive power of finite-state machines and their associated regular languages is limited. In this chapter, we are going to explore the more powerful computational model of *Turing machines* and their associated *recursively enumerable languages*. As it turns out, the expressiveness of this model is the same as that of a couple of other computational models, each of which may therefore serve as a substitute for Turing machines. In fact, its power is even equivalent to that of general purpose programming languages; there is no more powerful computational model known today. The two models *finite-state machines* and *Turing machines* represent the end points of a spectrum of machine models; the *Chomsky hierarchy* provides a comprehensive view on the relationship between these and other machine models and their associated languages.

3.1. Turing Machines

In this section, we discuss a computational model that was introduced by the British mathematician/logician/computer scientist Alan Turing in 1936.

3.1.1. Basics

We start with the core definition of this section.

Turing-Maschine

Definition 16 (Turing Machine) A *Turing machine* M is a 7-tuple $(Q, \Gamma, \sqcup, \Sigma, \delta, q_0, F)$ with the following components:

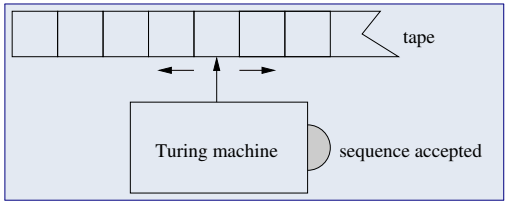


Figure 3.1.: A Turing Machine

Zustandsmenge
Zustand

- The *state set* Q, a fine set of elements called *states*.

Bandalphabet
Bandsymbol

- An *tape alphabet* Γ, an alphabet whose elements we call *tape symbols*.

Leersymbol

- The *blank symbol* $\sqcup \in \Gamma$, an element of Γ.

Eingabealphabet
Eingabesymbol

- An *input alphabet* $\Sigma \subseteq \Gamma \backslash \{\sqcup\}$, a subset of the tape alphabet that does not include the blank symbol; we call the elements of this alphabet *input symbols*.

Überführungsfunktion

- The *transition function* $\delta : Q \times \Gamma \to_p Q \times \Gamma \times \{\text{'L'}, \text{'R'}\}$, a partial function that takes a state and a tape symbol as an argument and may return a state, a tape symbol, and a "direction" 'L(eft)' or 'R(ight)'.

Anfangszustand

- The *start state* $q_0 \in Q$, one of the elements of Q.

akzeptierender Zustand
Endzustand

- A set of *accepting states* (also called *final states*) $F \subseteq Q$, a subset of Q.

Turing-Maschine
(Turing-)Maschine

We will frequently call a *Turing machine* just a *machine*.

Eingabe

Informally, the behavior of a Turing machine is as depicted in Figure 3.1. The machine operates on a tape of infinite length which in its initial part holds the *input*, a finite word over the input alphabet; the rest of the tape is filled with the blank symbol $\sqcup$. In the course of the execution, the tape can be overwritten with arbitrary symbols from the tape alphabet.

Lese-/Schreibkopf

In more detail, the machine has a *tape head* which can be used to read and write the symbol on the tape over which it is currently positioned; initially this is the first symbol of the tape. Starting from its start state, the machine executes a sequence of steps by repeated application of the transition function $\delta(q,a) = (q',a',d)$, which means that in its current state q it reads the symbol a under the tape head and (simultaneously)

- switches to its next state q',
- overwrites a with a symbol a', and
- moves the tape head into direction d, i.e., one symbol to the left or to the right.

The execution terminates if the Turing machine encounters a combination of state q and input symbol a for which it cannot make a further step. If q is an accepting state, the input is accepted.

One crucial difference to a finite-state machine is therefore that a Turing machine may not only read but also write the content of the tape and move over the tape in an arbitrary fashion; the tape is therefore not only an input medium but also becomes an output medium for the final result, as well as a storage medium for intermediate results.

Another crucial difference is that, for a given input word, the number of steps of the execution of a Turing machine is not bounded by the length of the word. The machine may perform arbitrarily many steps until it terminates; it may even not terminate at all, i.e., run forever. This seemingly disadvantage is actually the source of the additional power of a Turing machine compared to a finite-state machine: there is no trivial constraint (like the size of the input) that limits the number of steps the Turing machine may take for its execution.

Example 13 (Turing Machine) Take the Turing machine defined in Figure 3.2. As for automata, the transition function can be defined by a table which depicts, for every combination of state and tape symbol, the triple of successor state, output symbol, and tape head direction; if no entry is given, the Turing machine terminates for that combination of state and tape symbol. A state transition graph may be useful to give an overall impression of the behavior of the Turing machine, but does not entirely define it (unless also the output symbol and the the tape head direction are included, which however makes the diagram clumsy).

This Turing machine accepts every word of form 0^n1^n, i.e. all words of even length whose first half consists only of 0 and whose second half consists only of 1. The machine operates by replacing the left-most occurrence of 0 by X, moving the tape head right to the first occurrence of 1, moving the tape head again left to the first occurrence of X and then one position to the right. Then the execution is repeated in a loop. If the input word is of form 0^n1^n, the machine will terminate in an accepting state with tape content X^nY^n. □

The concept of a Turing machine can be generalized in various ways:

1. it may have a tape which is *infinite in both directions*;
2. it may have *multiple tapes*;

$M = (Q, \Gamma, \sqcup, \Sigma, \delta, q_0, F)$
$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$\Gamma = \{\sqcup, 0, 1, X, Y\}$
$\Sigma = \{0, 1\}$
$F = \{q_4\}$

δ	$\sqcup$	0	1	X	Y
q_0	–	(q_1, X, R)	–	–	(q_3, Y, R)
q_1	–	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)
q_2	–	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)
q_3	$(q_4, \sqcup, R)$	–	–	–	(q_3, Y, R)
q_4	–	–	–	–	–

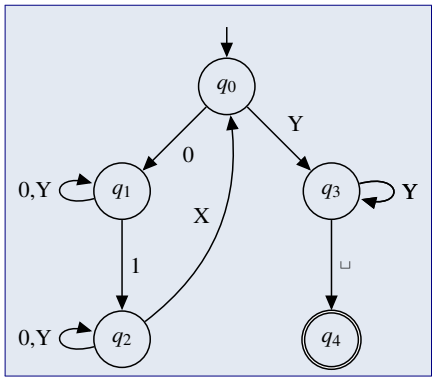


Figure 3.2.: A Turing Machine

3. it may be *nondeterministic*.

None of these extensions increases the expressive power of the concept, since they all can be simulated by the core form of a deterministic Turing machine with a single tape that is infinite in only one direction. We only show the most interesting case.

Theorem 7 (Nondeterministic Turing Machine) Let M be a nondeterministic Turing machine. Then there exists a deterministic Turing machine M' which accepts the same words as M.

Proof Let $M = (Q, \Gamma, \sqcup, \Sigma, \delta, q_0, F)$ be a nondeterministic Turing machine, i.e. a Turing machine with transition function

$$\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{\text{'L'}, \text{'R'}\})$$

such that $\delta(q, a)$ describes a set of possible successor situations. We then define

$$r := \max\{|\delta(q, a)| \mid q \in Q \wedge a \in \Gamma\}$$

i.e., r is the maximum number of "choices" that M can make in any situation.

We construct a deterministic Turing machine M' with three tapes (which can be subsequently transformed to a single tape Turing machine):

- Tape 1 (which will be only read) holds the input of the tape of M.
- On Tape 2, all finite sequences of values $1, \ldots, r$ are generated (first the one element sequences, then the two element sequences, and so on).
- Tape 3 is the working tape of M'.

Then M' mimics the execution of M by iterating the following process:

1. M' copies the input from Tape 1 to Tape 3.
2. M' generates the next sequence $s = s_1 s_2 \ldots s_n$ on Tape 2.
3. M' starts execution on Tape 3 by performing at most n execution steps. In the i-th step, M' applies the transition function δ of M and selects from the resulting set the element numbered s_i.

4. If this leads to a terminating state in F, M' accepts the input word and the process terminates; otherwise the process is repeated.

Our machine M' thus systematically simulates in a deterministic fashion all possible executions of the nondeterministic machine M; if there is one execution of M that accepts the input, also M' detects this execution and accepts its input. If there is no such execution, M' runs forever (i.e., does not accept its input). □

3.1.2. Recognizing Languages

We are now going to investigate the relationship between Turing machines and languages.

Definition 17 (Turing Machine Language) Let $M = (Q, \Gamma, \sqcup, \Sigma, \delta, q_0, F)$ be a Turing machine.

Konfiguration

- A *configuration* of M is a triple $(a_1 \dots a_k, q, a_{k+1} \dots a_m)$ (simply written as $a_1 \dots a_k\ q\ a_{k+1} \dots a_m$) with state q and tape symbols $\{a_1, \dots, a_k, a_{k+1}, \dots, a_m\} \subseteq \Gamma$.

  A configuration represents a situation in the execution of M: q is the current state of M, $a_1 \dots a_k$ represent the portion of the tape left to the tape head, a_{k+1} represents the symbol under the head, and $a_{k+2} \dots a_m$ represents the portion right to the head.

Zug

- The *move* relation $\vdash$ is a binary relation on configurations such that

$$a_1 \dots a_k\ q\ a_{k+1} \dots a_m \vdash b_1 \dots b_l\ p\ b_{l+1} \dots b_m$$

  holds if and only if $a_i = b_i$ for all $i \neq k+1$ and one of the following holds:

  $l = k+1$ and $\delta(q, a_{k+1}) = (p, b_l, R)$, or

  $l = k-1$ and $\delta(q, a_{k+1}) = (p, b_{l+2}, L)$.

  In other words, $c_1 \vdash c_2$ holds if, when M is in the situation described by configuration c_1, it can make a transition to the situation described by configuration c_2.

  We then define

$$\begin{aligned} c_1 \vdash^0 c_2 &:\Leftrightarrow c_1 = c_2 \\ c_1 \vdash^{i+1} c_2 &:\Leftrightarrow \exists c : c_1 \vdash^i c \wedge c \vdash c_2 \\ c_1 \vdash^* c_2 &:\Leftrightarrow \exists i \in \mathbb{N} : c_1 \vdash^i c_2 \end{aligned}$$

Thus $c_1 \vdash^i c_2$ holds if configuration c_1 can be transformed to configuration c_2 in i moves and $c_1 \vdash^* c_2$ holds if c_1 can be transformed to c_2 in an arbitrary number of moves.

die Sprache einer Turing-Maschine

- The *Turing machine language* $L(M)$ of M is the set of all inputs that drive M from its initial configuration to a configuration with an accepting state such that from this configuration no further move is possible:

$$L(M) := \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} \exists a, b \in \Gamma^*, q \in Q : q_0\ w \vdash^*\ a\ q\ b \wedge q \in F \\ \wedge\ \neg \exists a', b' \in \Gamma^*, q' \in Q : a\ q\ b\ \vdash\ a'\ q'\ b' \end{array} \right\}$$

Definition 18 (Recursively Enumerable and Recursive Languages)

rekursiv aufzählbare Sprache

1. L is a *recursively enumerable language* if there exists a Turing machine M such that $L = L(M)$.

rekursive Sprache

2. L is a *recursive language* if there exists a Turing machine M such that $L = L(M)$ and M terminates for every possible input.

erkennen

In both cases we also say that M *recognizes* L.

From the definition, every recursive language is recursively enumerable, but not necessarily vice versa. The Turing machine whose existence is required for a recursively enumerable language L need not terminate for all inputs that are not words of L. We will see in Chapter 4 that there indeed exist languages that are recursively enumerable but not recursive.

The exact relationship between recursively enumerable and recursive languages is captured by the following theorem.

Theorem 8 (Recursive Language) A language L is recursive if and only if both L and its complement $\overline{L}$ are recursively enumerable.

Proof We show both sides of the theorem.

```
function M'(w):
    answer ← M(w)
    case answer of
        yes: return no
        no: return yes
    end case
end function
```

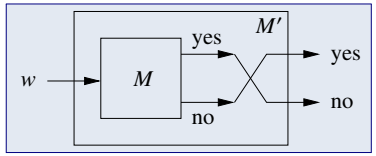


Figure 3.3.: Proof ⇒ of Theorem 8

⇒ Let L be a recursive language. We show that both L and $\overline{L}$ are recursively enumerable. Since by definition L is recursively enumerable, it remains to be shown that also $\overline{L}$ is recursively enumerable.

Since L is recursive, there exists a Turing machine M such that M halts for every input w: if $w \in L$, then M accepts w (we write $M(w) = \text{yes}$); if $w \notin L$, then M does not accept w (we write $M(w) = \text{no}$). With the help of M, we can construct the machine M' whose behavior is sketched as a program function and illustrated by a diagram in Figure 3.3.

We have by construction, for every input w, $M'(w) = \text{yes}$, if and only if $M(w) = \text{no}$, i.e., $w \notin L$. Thus $L(M') = \overline{L}$, therefore $\overline{L}$ is recursively enumerable.

⇐ Let language L be such that both L and $\overline{L}$ are recursively enumerable. We show that L is recursive.

Since L is recursively enumerable, there exists a Turing machine M such that $L = L(M)$ and M halts for every input $w \in L$ with answer $M(w) = \text{yes}$. Since $\overline{L}$ is recursively enumerable, there exists a Turing machine $\overline{M}$ such that $\overline{L} = L(\overline{M})$ and $\overline{M}$ halts for every input $w \in \overline{L}$ (i.e., $w \notin L$) with answer $\overline{M}(w) = \text{yes}$. With the help of M and $\overline{M}$, we can construct the machine M'' sketched in Figure 3.4.

In the execution of $M''(w)$, we simulate the parallel execution of M and $\overline{M}$: if $w \in L$, then $M(w)$ terminates with $M(w) = \text{yes}$ and therefore $M''(w)$ terminates with $M''(w) = \text{yes}$; if $w \notin L$, then $\overline{M}(w)$ terminates with $\overline{M}(w) = \text{yes}$ and therefore $M''(w)$ terminates with $M''(w) = \text{no}$. Thus M'' terminates for every input and $L(M'') = L$. Therefore L is recursive.

Please note that if $w \notin L$, the execution of $M(w)$ may not terminate (as indicated by a loop in the diagram); the correctness of $M''(w)$ must therefore not depend on the answer $M(w) = \text{no}$ (even if such an answer arises, M'' may ignore it, i.e., the corresponding branch of the parallel execution may loop forever). Correspondingly, if $w \notin \overline{L}$, the


```
function M''(w):
    parallel
        begin
            answer_1 ← M(w)
            if answer_1 = yes then
                return yes
            else
                loop forever
            end if
        end
        begin
            answer_2 ← M̄(w)
            if answer_2 = yes then
                return no
            else
                loop forever
            end if
        end
    end parallel
end function
```

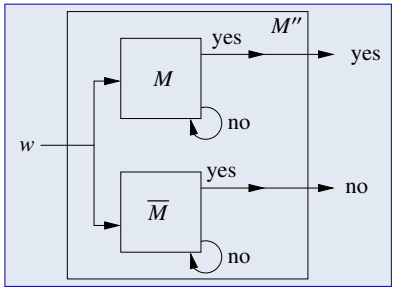


Figure 3.4.: Proof ⇐ of Theorem 8

execution of $\overline{M}(w)$ may not terminate such that the correctness of $M''(w)$ must not depend on the answer $\overline{M}(w) = \text{no}$.

In above proof, we have borrowed programming language constructs for the construction of Turing machines. We will justify in Section 3.2.2 this "abuse of notation".

Recursive languages are closed under the usual set operations.

Theorem 9 (Closure of Recursive Languages) Let L, L_1, L_2 be recursive languages. Then also

- the complement $\overline{L}$,
- the union $L_1 \cup L_2$,
- the intersection $L_1 \cap L_2$

are recursive languages.

Proof By construction of the corresponding Turing machines (exercise). □

3.1.3. Generating Languages

Certain versions of Turing machines may be used to generate languages.

Definition 19 (Enumerator, Generated Language)

- Let $M = (Q, \Gamma, \sqcup, \emptyset, \delta, q_0, F)$ be a Turing machine where the tape alphabet Γ contains a special symbol #. Then M is called an *enumerator* if it has (additionally to its *working tape*) an *output tape* on which M moves its tape head only to the right (or lets it stay in its current position) and writes only symbols different from the blank symbol $\sqcup$.

Aufzähler

Arbeitsband

Ausgabeband

- The *generated language* $Gen(M)$ of enumerator M is the set of all words that M, after starting with all tapes in an empty state (i.e., filled with the $\sqcup$ symbol), eventually writes on the output tape where the end of each word is marked by the special symbol # (which itself does not belong to the word any more).

erzeugte Sprache

It should be noted that an enumerator M may run forever and that the generated language $Gen(M)$ may thus be infinite. For instance, if M writes on the output tape the sequence

$$\#\,0\,\#\,0\,0\,\#\,0\,0\,0\,\#\,0\,0\,0\,0\,\#\,\ldots$$

then $Gen(M) = L(0^*)$.

Then we have the following theorem which justifies the name "(recursively) *enumerable*" for any language that is accepted by a Turing machine.

aufzählbar

Theorem 10 (Generated Langues) A language L is recursively enumerable if and only if there exists some enumerator M such that $L = Gen(M)$.

Proof We show both parts of the theorem.

$\Rightarrow$ Let L be a recursively enumerable language. We show that there exists some enumerator M such that $L = Gen(M)$.

Since L is recursively enumerable, there exists some Turing machine M' such that $L = L(M')$. We use M' to build the enumerator M which works as follows (see Figure 3.5): M produces on its working tape every pair (m,n) of natural numbers $m, n \in \mathbb{N}$ (encoded in some form):


```
procedure M:
    loop
        produce next (m,n) on working tape
        if M'(w_m) = yes in at most n steps then
            write w_m to output tape
        end if
    end loop
end procedure
```

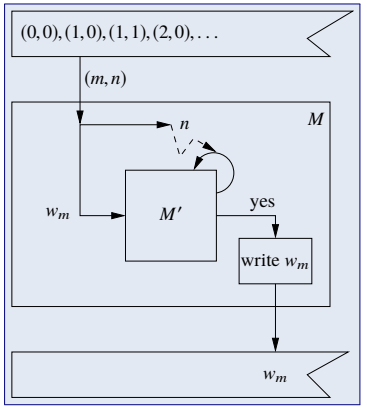


Figure 3.5.: Proof $\Rightarrow$ of Theorem 10

$$(0,0),(1,0),(1,1),(2,0),(2,1),(2,2),(3,0),\ldots$$

Whenever a new pair (m,n) has been produced, M constructs the m-th word w_m over the input alphabet Σ of M (this is easily possible, because all words of Σ can be ordered by first listing the empty word ε, then all words of length 1, then all words of length 2, etc). Then M' performs at most n steps of the execution of $M(w_m)$. If these steps lead to the acceptance of w_m (i.e., $M(w_m) = \text{yes}$) then M' writes w_m to the output tape. Afterwards, M' proceeds with the next pair (m,n).

Then $Gen(M') = L(M)$, i.e., for every word w, $w \in Gen(M') \Leftrightarrow w \in L(M)$:

- $\Rightarrow$ If $w \in Gen(M')$, i.e., M' writes w to the output tape, then w has been accepted by M, i.e., $w \in L(M)$.
- $\Leftarrow$ If $w \in L(M)$, then there exists some position m such that $w = w_m$ and some number n of steps in which w is accepted by M. This pair (m,n) is eventually produced by M' which thus eventually writes w on the output tape, i.e. $w \in Gen(M')$.

$\Leftarrow$ Let L be such that $L = Gen(M)$ for some enumerator M. We show that there exists a Turing machine M' such that $L = L(M')$.

We use enumerator M to build M' which works as follows (see Figure 3.6): given input w, M' invokes M to produce, word for word, the sequence of all words in $Gen(M)$. If w eventually appears in this sequence, M' accepts w, i.e. $M'(w) = \text{yes}$. Otherwise, M' may not terminate (if M produces infinitely many words).

```
function M'(w):
    while M is not terminated do
        M writes next word w' to output tape
        if w = w' then
            return yes
        end if
    end while
    return no
end function
```

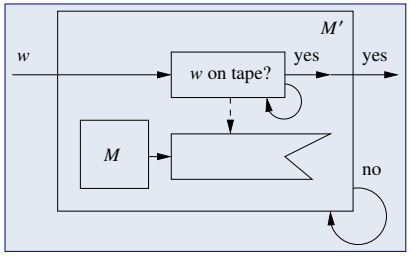


Figure 3.6.: Proof ⇐ of Theorem 10

Then $L(M') = Gen(M)$, i.e., for every word w, $w \in L(M') \Leftrightarrow w \in Gen(M)$:

⇒ If $w \in L(M')$, i.e., $M'(w) =$ yes, then w appears in the sequence generated by M, i.e. $w \in Gen(M)$.

⇐ If $w \in Gen(M)$, i.e., w eventually appears in the sequence generated by M, then input w is accepted by M, i.e. $w \in L(M')$. □

Theorem 10 shows us that generating a recursively enumerable language is exactly as difficult as recognizing such a language. For any recursively enumerable language, we may therefore subsequently not only assume an accepting Turing machine but also a generating one.

3.1.4. Computing Functions

In this section, we will investigate how Turing machines can be used to compute mathematical functions. First we recall some basic notions.

Definition 20 (Functions) Let $f \subseteq A \times B$ be a relation between sets A and B (i.e., a set of pairs (a, b) with $a \in A$ and $b \in B$).

(totale) Funktion

1. Relation f is a *function* from A to B, written $f : A \to B$, if for every value $a \in A$ there is *exactly one* value $b \in B$ such that $(a, b) \in f$.

partielle Funktion

2. Relation f is a *partial function* from A to B, written $f : A \to_p B$, if for every value $a \in A$ there is *at most one* value $b \in B$ such that $(a, b) \in f$.

Definitionsbereich

3. For a partial function f, we define the *domain* of f as

$$domain(f) := \{a \mid \exists b : (a, b) \in f\}$$

i.e., as the set of all elements a such that there exists an element b with $(a, b) \in f$.

Wertebereich

4. For a partial function f, we define the *range* of f as

$$range(f) := \{b \mid \exists a : (a, b) \in f\}$$

i.e., as the set of all elements b such that there exists an element a with $(a, b) \in f$.

(Funktions-)Ergebnis (Funktions-)Argument

5. For a partial function f, if $(a, b) \in f$, we write $f(a)$ to denote the *result* b of the application of f to *argument* a.

Please note that every total function $f : A \to B$ is also a partial function $f : A \to_p B$ with $domain(f) = A$, and every partial function $f : A \to_p B$ is a total function $f : domain(A) \to B$.

The concept of the "computability" of a mathematical function is now formalized on the basis of Turing machines.

Definition 21 (Turing Computability) Let Σ and Γ be alphabets that do not contain the symbol $\sqcup$ and $f : \Sigma^* \to_p \Gamma^*$ be a partial function from words over Σ to words over Γ. Then f is called *Turing computable* if there exists a Turing machine M such that

(Turing-)berechenbar

- for input w (i.e. initial tape content $w\sqcup\ldots$), M terminates in an accepting state if and only if $w \in domain(f)$;
- for input w, M terminates in an accepting state with output w' (i.e. final tape content $w'\sqcup\ldots$) if and only if $w' = f(w)$.

For a function to be (Turing) computable, the corresponding Turing machine needs to terminate for all arguments from the domain of the function with the function result; for all arguments outside the domain, the machine must not terminate.

A simple argument shows there indeed exist functions from words over Σ to words over Γ that are *not* computable by a Turing machine: every Turing machine can be formalized by a finite description (say a bit string) and all bit strings can be ordered in a (of course infinite) sequence; thus there is a one to one correspondence between the set of natural numbers (positions in the sequence) and Turing machines (descriptions in the sequence); thus the set of all Turing machines is *countably infinite*. However, by application of Cantor's *diagonal argument*[1] (which

abzählbar unendlich

[1] http://en.wikipedia.org/wiki/Cantor's_diagonal_argument

will also play a role in Chapter 4), one can show that the set of all functions from Σ^* to Γ^* is more than countably infinite (the set of all word functions cannot be listed in a sequence indexed by natural numbers). So not for every word function there can exist a Turing machine that computes this function, there are simply too many of them!

Nevertheless most functions that are in "daily use" are computable, e.g., the usual arithmetic functions over (word representations of) the natural numbers.

Example 14 (Turing Computability) We show that the natural number difference

$$m \ominus n := \begin{cases} m - n & \text{if } m \geq n \\ 0 & \text{else} \end{cases}$$

is Turing computable, using for the natural number $n \in \mathbb{N}$ the *unary* representation

$$\underbrace{000\ldots0}_{n \text{ times}} \in L(0^*)$$

i.e., for input $00\sqcup0$ $(= 2 \ominus 1)$, the output shall be 0 $(= 1)$.

The Turing machine M that computes $\ominus$ operates as follows (see Figure 3.7):

- In start state q_0, the leading 0 is replaced by $\sqcup$.
- In state q_1, M searches for the next $\sqcup$ and replaces it by a 1.
- In state q_2, M searches for the next 0 and replaces it by 1, then moves left.
- In state q_3, M searches for the previous $\sqcup$, moves right and starts from the beginning.
- In state q_4, M has found a $\sqcup$ instead of a 0; it therefore replaces all previous 1 by $\sqcup$.
- In state q_5, n is (has become) 0; the rest of the tape is erased.
- In state q_6, the computation successfully terminates.

Thus the computation of M on input $00\sqcup0$ $(= 2 \ominus 1)$ yields the following sequence of moves

$$\begin{aligned} &q_000\sqcup0 \vdash \sqcup q_10\sqcup0 \vdash \sqcup0q_1\sqcup0 \vdash \sqcup01q_20 \\ &\vdash \sqcup0q_311 \vdash \sqcup q_3011 \vdash q_3\sqcup011 \vdash \sqcup q_0011 \\ &\vdash \sqcup\sqcup q_111 \vdash \sqcup\sqcup1q_21 \vdash \sqcup\sqcup11q_2 \vdash \sqcup\sqcup1q_41 \\ &\vdash \sqcup\sqcup q_41 \vdash \sqcup q_4 \vdash \sqcup0q_6. \end{aligned}$$

while input $0\sqcup00$ $(= 1 \ominus 2)$ results in

$M = (Q, \Gamma, \sqcup, \Sigma, \delta, q_0, F)$
$Q = \{q_0, \ldots, q_6\}$
$\Sigma = \{0\}$
$\Gamma = \{0, 1, \sqcup\}$
$F = \{q_6\}$

δ	0	1	$\sqcup$
q_0	$(q_1, \sqcup, R)$	$(q_5, \sqcup, R)$	$(q_5, \sqcup, R)$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$(q_2, 1, R)$
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	$(q_4, \sqcup, L)$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	$(q_0, \sqcup, R)$
q_4	$(q_4, 0, L)$	$(q_4, \sqcup, L)$	$(q_6, 0, R)$
q_5	$(q_5, \sqcup, R)$	$(q_5, \sqcup, R)$	$(q_6, \sqcup, R)$
q_6	–	–	–

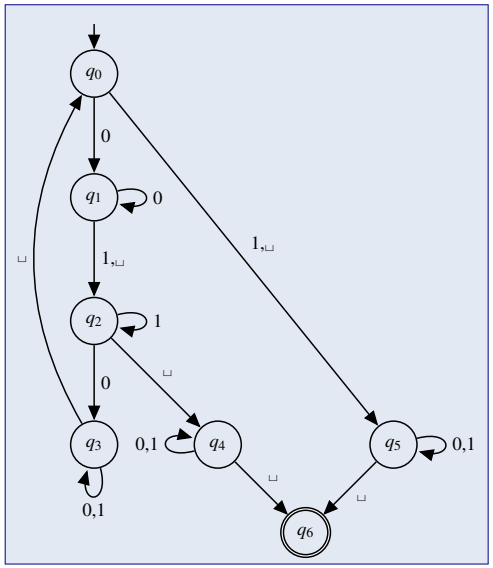


Figure 3.7.: Computing $m \ominus n$

```
function M'(a, b):
    b' ← M(a)
    if b' = b then
        return yes
    else
        return no
    end if
end function
```

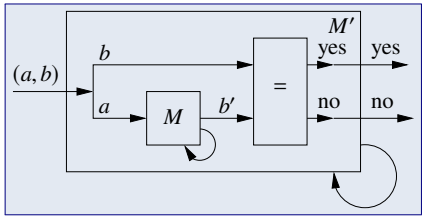


Figure 3.8.: Proof ⇒ of Theorem 11

$q_0 0\sqcup 00 \vdash \sqcup q_1 \sqcup 00 \vdash \sqcup 1 q_2 00 \vdash \sqcup q_3 110$
$\vdash q_3 \sqcup 110 \vdash \sqcup q_0 110 \vdash \sqcup\sqcup q_5 10 \vdash \sqcup\sqcup\sqcup q_5 0$
$\vdash \sqcup\sqcup\sqcup q_5 \vdash \sqcup\sqcup\sqcup\sqcup q_6.$

For $m > n$ the computation of $m \ominus n$ generates leading blanks in the output that still need to be removed; we leave the corresponding modification of the Turing machine as an exercise. □

The following theorem provides a relationship between Turing computable functions and Turing recognizable languages.

Theorem 11 (Turing Computability) A partial function $f : \Sigma^* \to_p \Gamma^*$ is Turing computable, if and only if the language

$$L_f := \{(a, b) \in \Sigma^* \times \Gamma^* \mid a \in domain(f) \wedge b = f(a)\}$$

is recursively enumerable.

Proof We show both directions of the theorem.

⇒ Since $f : \Sigma^* \to_p \Gamma^*$ is Turing computable, there exists a Turing machine M which computes f. To show that L_f is recursively enumerable, we use M to construct the following Turing machine M' with $L(M') = L_f$ (see Figure 3.8): M' takes the input (a, b) and invokes M on a: if M terminates in an accepting state with output b, M' accepts the input, otherwise it does not (if M does not terminate, then also M' does not terminate).

⇐ Since L_f is recursively enumerable, there exists by Theorem 10 an enumerator M with $Gen(M) = L_f$. We use M to construct the following Turing machine M' which computes f


```
function M'(a):
    while M is not terminated do
        M writes pair (a', b') to tape
        if a = a' then
            return b'
        end if
    end while
    loop forever
end function
```

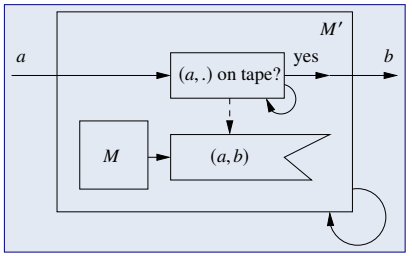


Figure 3.9.: Proof ⇐ of Theorem 11

(see Figure 3.9): M' invokes M to generate all pairs (a, b) with $a \in domain(f) \wedge b = f(a)$. If the argument a of M eventually appears in this enumeration, M returns the corresponding result b; otherwise M does not terminate. □

As Turing machines provide a mean of computing mathematical functions, we will now address the question whether there exist other (more powerful) notions of computability of those than provided by Turing machines.

3.1.5. The Church-Turing Thesis

In computer science, *algorithms* plays a central role: intuitively, an algorithm denotes a description of a procedure for carrying out some calculation or task; this description is such precise that it can be ultimately performed by a machine. A core question of computer science is to find, for a given computational problem, an algorithm that solves this problem, respectively to show that no such algorithm does exist. Unfortunately, the notion of an "algorithm" is actually *only* an intuitive one; it is not sufficient to perform mathematical proofs about the (non-)existence of algorithms for given problems. Algorithmus

In 1936, Alonzo Church and Alan Turing therefore came up with two different approaches to replace the intuitive notion of algorithms by formal computational models:

- Church invented the *lambda calculus*, a notation to describe mechanically computable functions (see Section 3.2.4); Lambda-Kalkül
- Turing described a formal concept of *machines* which we today call *Turing machines*.

While both concepts looked fundamentally different, Church and Turing could prove that their computational power was exactly the same, i.e., that the set of mathematical functions that can be described by the lambda calculus is the same as the set of functions computable by Turing

machines. This coincidence let Church and Turing conjecture, that their formal models already capture *everything* that is algorithmically solvable.

Thesis 1 (Church-Turing Thesis) Every problem that is solvable by an algorithm (in an intuitive sense) is solvable by a Turing machine. Consequently, the set of intuitively computable functions is identical with the set of Turing computable functions.

This statement is not a (provable) theorem but only an (unprovable) *thesis*, because of exactly the reason that "algorithm" is not a formal notion. However, the thesis has been subsequently empirically validated numerous times by showing that the power of many other (seemingly very different) computational models is exactly the same as the one of Turing machines; in fact, no more powerful computational model is known.

3.2. Turing Complete Computational Models

In this section, we will sketch various computational models that have been shown to be *Turing complete* (also called *universal*), i.e., to have the same computational power as Turing machines. We will also discuss some interesting restricted versions that are not Turing complete and outline their relationships to the Turing complete models.

Turing-vollständig
universell

3.2.1. Random Access Machines

While Turing machines are simple, they are conceptually very different from what we call today a "computer". In this section, we will discuss a machine model that is much closer to those machines with which we operate day by day.

Registermaschine

Definition 22 (Random Access Machine) A *random access machine* (*RAM*) is a machine that consists of the following components (see Figure 3.10):

- an infinite input tape I (whose cells can hold natural numbers of arbitrary size) with a read head position $i \in \mathbb{N}$,
- an infinite output tape O (whose cells can hold natural numbers of arbitrary size) with a write head position $o \in \mathbb{N}$,

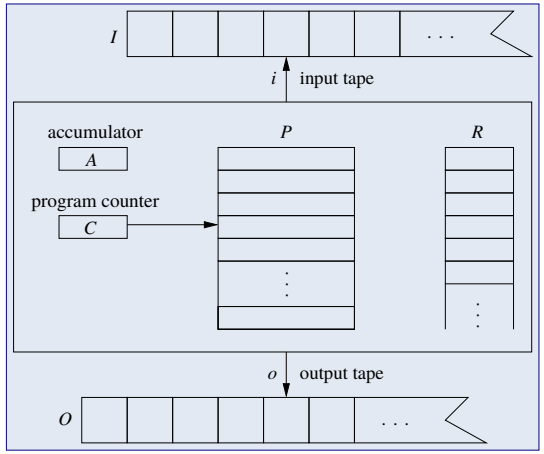


Figure 3.10.: A Random Access Machine

- an accumulator A which can hold a natural number of arbitrary size,
- a program counter C which can hold a natural number of arbitrary size,
- a program consisting of a finite number of instructions $P[1], \ldots, P[m]$, and
- a memory consisting of a countably infinite sequence of registers $R[1], R[2], \ldots$, each of which can hold a natural number of arbitrary size.

Initially, the positions i and o are 0, A is 0, C is 1, and all registers $R[1], R[2], \ldots$ are 0. In every step of its execution, the machine reads the instruction $P[C]$, increments C by 1, and then performs the action indicated by the instruction (the action is undefined if the instruction is executed on an invalid argument):

Instruction	Description	Action
`IN`	Read value from input tape into accumulator	$A := I[i]; i := i + 1$
`OUT`	Write value from accumulator to output tape	$O[o] := A; o := o + 1$
`LOAD #n`	Load constant n into accumulator	$A := n$
`LOAD n`	Load content of register n into accumulator	$A := R[n]$
`LOAD (n)`	Load content of register referenced by register n	$A := R[R[n]]$

START:	`LOAD #1`	$A := 1$
	`STORE 1`	$R[1] := A$
READ:	`LOAD 1`	$A := R[1]$
	`ADD #1`	$A := A + 1$
	`STORE 1`	$R[1] := A$
	`IN`	$A := I[i]; i := i + 1$
	`BEQ 0,WRITE`	if $A = 0$ then $C :=$ WRITE
	`STORE (1)`	$R[R[1]] := A$
	`JUMP READ`	$C :=$ READ
WRITE:	`LOAD 1`	$A := R[1]$
	`SUB #1`	$A := A - 1$
	`STORE 1`	$R[1] := A$
	`BEQ 1,HALT`	if $A = 1$ then $C :=$ HALT
	`LOAD (1)`	$A := R[R[1]]$
	`OUT`	$O[o] := A; o := o + 1$
	`JUMP WRITE`	$C :=$ WRITE
HALT:	`JUMP 0`	$C := 0$

Figure 3.11.: Execution of RAM

`STORE n`	Store content of accumulator into register n	$R[n] := A$
`STORE (n)`	Store content into register referenced by register n	$R[R[n]] := A$
`ADD #n`	Increment content of accumulator by constant	$A := A + n$
`SUB #n`	Decrement content of accumulator by constant	$A := \max\{0, A - n\}$
`JUMP n`	Unconditional jump to instruction n	$C := n$
`BEQ i,n`	Conditional jump to instruction n	if $A = i$ then $C := n$

The execution terminates if C gets value 0.

Example 15 (Execution of RAM) We construct a RAM that reads from the input tape a sequence of natural numbers terminated by 0 and writes the sequence (excluding the 0) in inverted order to the output tape. The program of the RAM is depicted in Figure 3.11 (for readability, jump addresses are replaced by symbolic labels).

The RAM uses the memory as a stack which is indexed by the register $R[1]$. In the *read* phase of its execution, the RAM writes into $R[2], R[3], \ldots$ the content of the input tape until it encounters 0. The RAM switches then to the *write* phase where it writes the values $\ldots, R[3], R[2]$ to the output tape, until $R[1] = 1$.

Finally, the RAM jumps to address 0, which terminates the execution. □

The model of random access machines is Turing complete.

Theorem 12 (Turing Machines and RAMs) Every Turing machine can be simulated by a RAM and vice versa.

Proof We sketch the proofs of both parts of the theorem.

⇒ The RAM uses some registers $R[1], \ldots, R[c-1]$ for its own purposes, stores in register $R[c]$ the position of the tape head of the Turing machine and (applying the facilities for *indirect addressing* provided by `LOAD(n)` and `STORE(n)`) uses the registers $R[c+1], R[c+2], \ldots$ as a "virtual tape" to simulate the tape of the Turing machine. First, the RAM copies the input from the input tape into its virtual tape, then it mimics the execution of the Turing machine on the virtual tape. When the simulated Turing machine terminates, the content of the virtual tape is copied to the output tape.

⇐ The Turing machine uses 5 tapes to simulate the RAM:

- Tape 1 represents the input tape of the RAM.
- Tape 2 represents the output tape of the RAM.
- Tape 3 holds a representation of that part of the memory that has been written by the simulation of the RAM.
- Tape 4 holds a representation of the accumulator of the RAM.
- Tape 5 serves as a working tape.

Tape 3 holds a sequence of (address,contents) pairs that represent those registers of the RAM that have been written during the simulation (the contents of all other registers can be assumed to hold 0). Every instruction of the RAM is simulated by a sequence of steps of the Turing machine which reads respectively writes Tape 1 and 2 and updates on Tape 3 and 4 the tape representations of the contents of the memory and the accumulator. □

A RAM is quite similar to an actual computer; the main difference is that in a computer the program is stored in the same memory as the data on which it operates. One can actually

Registermaschine mit gespeichertem Programm

generate a variant of the RAM model, the *random access stored program machine* (*RASP*) which gives up the distinction between P and R (i.e. the program is stored in R and can be modified during the execution of the RASP). One can then show, that a RAM can still simulate the execution of the RASP by a program that serves as an *interpreter* for the RASP instructions (like a *microprogram* on a modern microprocessor may interpret the instructions of the processor's machine language).

One may even remove from a RASP the instructions with indirect addressing `LOAD(n)` and `STORE(n)`. Even with this restriction, it can be shown that a RASP is able to simulate a RAM: every instruction `LOAD(n)` respectively `STORE(n)` is replaced by a dummy instruction which is at runtime overwritten by its *direct addressing* counterpart `LOAD m` respectively `STORE m` with calculated register number m. Thus also this restricted form of a RASP is Turing complete.

3.2.2. Loop and While Programs

In this section, we turn to computational models that resemble the programs written in structured programming languages.

Loop-Programm

Definition 23 (Loop Program Syntax) A *loop program* is an element of the set P which is inductively defined by the following grammar:

$$P \ ::= x_i := 0 \mid x_i := x_j + 1 \mid x_i := x_j - 1 \mid P; P \mid \textbf{loop } x_i \textbf{ do } P \textbf{ end}.$$

where x_i and x_j denote some variables from the set $\{x_0, x_1, x_2, \ldots\}$ of program variables.

A loop program thus includes most of the core commands of real programming languages: assignments, sequences, and bounded iteration (the intuitive meaning of the **loop** construct is that P is repeated as often as the initial value of x_i before entering the loop says).

The meaning of a loop program is determined the memory state after its execution.

Semantik

Definition 24 (Loop Program Semantics) Given memory $m : \mathbb{N} \to \mathbb{N}$, i.e., a mapping of variable numbers to variable contents, and loop program P, the *semantics* $[\![P]\!](m)$ is inductively defined as the final memory that results from the execution of P with start memory m:

$$\begin{aligned}
[\![x_i := 0]\!](m) &:= m[i \leftarrow 0] \\
[\![x_i := x_j + 1]\!](m) &:= m[i \leftarrow m(j) + 1] \\
[\![x_i := x_j - 1]\!](m) &:= m[i \leftarrow \max\{0, m(j) - 1\}] \\
[\![P_1; P_2]\!](m) &:= [\![P_2]\!]([\![P_1]\!](m)) \\
[\![\textbf{loop } x_i \textbf{ do } P \textbf{ end}]\!](m) &:= [\![P]\!]^{m(i)}(m)
\end{aligned}$$

Here $m[i \leftarrow n]$ denotes the memory resulting from m by updating variable x_i with value n; $[\![P]\!]^n(m)$ is the memory resulting from the n-fold repetition of program P in memory m:

$$\begin{aligned}
[\![P]\!]^0(m) &:= m \\
[\![P]\!]^{n+1}(m) &:= [\![P]\!]([\![P]\!]^n(m))
\end{aligned}$$

We can easily copy the content of variable x_j into variable x_i by the loop program

$$x_i := x_j + 1; x_i := x_i - 1$$

which we will abbreviate as $x_i := x_j$. Likewise, we abbreviate by $x_i := n$ the loop program

$$x_i := 0; x_i := x_i + 1; x_i := x_i + 1; \ldots; x_i := x_i + 1$$

which sets variable x_i to natural number n. Furthermore, we can simulate a conditional statement **if** $x_i = 0$ **then** P_t **else** P_e **end** by the loop program

$$\begin{array}{l} x_t := 1; \textbf{loop } x_i \textbf{ do } x_t := 0; \textbf{ end}; \\ x_e := 1; \textbf{loop } x_t \textbf{ do } x_e := 0; \textbf{ end}; \\ \textbf{loop } x_t \textbf{ do } P_t \textbf{ end}; \textbf{loop } x_e \textbf{ do } P_e \textbf{ end}; \end{array}$$

where x_t and x_e denote variables that are not in use in the rest of the program. If $x_i = 0$, then x_t is set to 1 and x_e is set to 0, else x_t is set to 0 and x_e is set to 1. Then P_t and P_e are executed as often as indicated by x_t and x_e. We will thus freely use also conditionals in loop programs.

Loop programs can compute functions over the natural numbers in the following sense.

Loop-berechenbar

Definition 25 (Loop Computability) A function $f : \mathbb{N}^n \to \mathbb{N}$ over the natural numbers is *loop computable* if there exists a loop program P such that for all $x_1, \ldots, x_n \in \mathbb{N}$ and memory $m : \mathbb{N} \to \mathbb{N}$ defined as

$$m(i) := \begin{cases} x_i & \text{if } 1 \le i \le n \\ 0 & \text{else} \end{cases}$$

we have $[\![P]\!](m)(0) = f(x_1, \ldots, x_n)$.

In other words, when started in a state where the variables $x_1, \ldots, x_n$ contain the arguments of f (and all other variables contain 0), the program terminates in a state where the variable x_0 holds the function result $f(x_1, \ldots, x_n)$.

Since loop programs only allow bounded iteration, i.e., the number of loop iterations must be known in advance, all loop programs terminate after a finite number of execution steps. Thus every loop computable function is total. We give some examples.

Example 16 (Loop Computability) We abbreviate by $x_0 := x_1 + x_2$ the following program which computes the sum of x_1 and x_2 (i.e., natural number addition is loop computable).

```
x0 := x1;
loop x2 do
  x0 := x0 + 1
end
```

Using this program, we can define the program $x_0 := x_1 \cdot x_2$ to compute the product of x_1 and x_2 (i.e., natural number multiplication is loop computable):

```
x0 := 0;
loop x2 do
  x0 := x0 + x1
end
```

Using this program, we can define the program $x_0 := x_1^{x_2}$ to compute the value of x_1 raised to the power of x_2 (i.e., natural number exponentiation is loop computable):

```
x0 := 1;
loop x2 do
  x0 := x0 · x1
end
```

□

The previous example shows that by nesting more and more loops arithmetic operations of higher degree can be implemented: above program $x_0 := x_1 \cdot x_2$ is actually an abbreviation for the doubly nested loop


```
x0 := 0;
loop x2 do
  x0 := x0;
  loop x1 do
    x0 := x0 + 1
  end
end
```

and the program $x_0 := x_1^{x_2}$ is actually an abbreviation for a triply nested loop.

We can extend the computational pattern described above beyond exponentiation by defining, for $a, b \in \mathbb{N}$ and $n \in \mathbb{N}\backslash\{0\}$, an operation $a \uparrow^n b$:

$$a \uparrow^n b := \begin{cases} a^b & \text{if } n = 1 \\ 1 & \text{if } b = 0 \\ a \uparrow^{n-1} (a \uparrow^n (b-1)) & \text{else} \end{cases}$$

Then

$$a \uparrow^1 b = a \cdot a \cdot \ldots \cdot a \ \ (b \text{ times})$$

i.e., $a \uparrow^1 b = a^b$, and

$$a \uparrow^2 b = a \uparrow^1 a \uparrow^1 \ldots \uparrow^1 a \ \ (b \text{ times})$$

i.e, $a \uparrow^2 b = a^{a^{\cdot^{\cdot^{a}}}}$ (b times), and further on

$$a \uparrow^3 b = a \uparrow^2 a \uparrow^2 \ldots \uparrow^2 a \ \ (b \text{ times})$$

for which no other well known notation exists any more. In general, the operation $a \uparrow^n b$ thus describes the b-fold repeated application of operation $\uparrow^{n-1}$ to a.

Then we can generalize the line of thought that started with the example above by the following theorem.

Theorem 13 (Function $a \uparrow^n b$ and Loops) Let $n \in \mathbb{N}$ and $f : \mathbb{N}^2 \to \mathbb{N}, f(a, b) := a \uparrow^n b$. Then

- f is loop computable, and
- every loop program computing f requires at least $n + 2$ nested loops.

For instance, this result (which we state without proof) claims that the computation of $a \uparrow^1 b = a^b$ requires three nested loops which corresponds with our observation above. However, from this positive result we can conclude another negative result.

Theorem 14 (Computability of $a \uparrow^n b$) The function $g : \mathbb{N}^3 \to \mathbb{N}, g(a,b,n) := a \uparrow^{n+1} b$ is not loop computable.

Proof Assume a loop program P which computes g and let n be the maximum number of nested loops in P. Then for input (a,b,n) the program cannot always compute the correct result $g(a,b,n) = a \uparrow^{n+1} b$, because this requires, by Theorem 13, at least $n+3$ loops. □

This result can also be understood in that, if n is increased, the numbers computed by $a \uparrow^n b$ become rapidly very big, in fact, too big to be computed by a finite number of bounded loops.

Ackermann-Funktion

Another function with a similar growth pattern is the *Ackermann function* defined as

$$\begin{aligned} ack(0,m) &:= m+1 \\ ack(n,0) &:= ack(n-1,1) \\ ack(n,m) &:= ack(n-1, ack(n,m-1)), \text{ if } n > 0 \wedge m > 0 \end{aligned}$$

where $ack(4,2)$ is a number with about 20,000 digits. In fact, since one can show that, for all $n,m \in \mathbb{N}$, $ack(n,m) = 2 \uparrow^{n-2} (m+3) - 3$, also the Ackermann function is not loop computable.

Since the power of bounded loops is apparently limited, we now extend the computational model by unbounded loops.

While-Programm

Definition 26 (While Program Syntax) A *while program* is an element of the set P which is inductively defined by the following grammar:

$$P \ ::= \ \ldots \text{ (as in Definition 23)} \mid \textbf{while } x_i \textbf{ do } P \textbf{ end}$$

The intuitive meaning of the **while** construct is that P is repeated as long as the value of x_i is not zero. If the value of x_i remains non zero forever, the program does not terminate. This intuition can be formalized as follows.

Definition 27 (While Program Semantics) Let the state m of a program be either a memory $m : \mathbb{N} \to \mathbb{N}$, or the value $m = \perp$ (*bottom*) indicating *non-termination*.

Bottom

We then define the *semantics* $[\![P]\!](m)$ of a while program P as the state that results from the execution of P when started in state m:

Semantik

- If the start state m indicates non-termination of a previously executed program, the whole program does not terminate:

$$[\![P]\!](m) := \begin{cases} \perp & \text{if } m = \perp \\ [\![P]\!]'(m) & \text{else} \end{cases}$$

- Given a memory m, for all commands that also appear in loop programs, the semantics remains unchanged:

$$[\![\ldots]\!]'(m) \ := \ \ldots \text{ (as in Definition 24)}$$

- The execution of a while loop may result in another memory or in non-termination:

$$[\![\textbf{while } x_i \textbf{ do } P \textbf{ end}]\!]'(m) := \begin{cases} \perp & \text{if } L_i(P,m) \\ [\![P]\!]^{T_i(P,m)}(m) & \text{else} \end{cases}$$

  The predicate $L_i(P,m)$ holds if, starting with memory m, by repeated application of the loop body P the loop variable x_i never gets zero. Otherwise $T_i(P,m)$ denotes the smallest number of iterations after which x_i becomes 0:

$$\begin{aligned} L_i(P,m) &:\Leftrightarrow \forall k \in \mathbb{N} : [\![P]\!]^k(m)(i) \neq 0 \\ T_i(P,m) &:= \min\{k \in \mathbb{N} \mid [\![P]\!]^k(m)(i) = 0\} \end{aligned}$$

More general versions of **while** loop conditions can be simulated by the restricted form presented above; e.g., we can consider the loop **while** $x_i < x_j$ **do** P **end** as an abbreviation for

$x_k := x_j - x_i;$
while x_k **do** $P; x_k := x_j - x_i;$ **end**

We will now generalize the notion of computability introduced in Definition 25.

```
function ack(n, m):
    if n = 0 then
        return m + 1
    else if m = 0 then
        return ack(n − 1, 1)
    else
        return ack(n − 1, ack(n, m − 1))
    end if
end function
```

```
function ack(x1, x2):
    push(x1); push(x2)
    while size() > 1 do
        x2 ← pop(); x1 ← pop()
        if x1 = 0 then
            push(x2 + 1)
        else if x2 = 0 then
            push(x1 − 1); push(1)
        else
            push(x1 − 1); push(x1); push(x2 − 1)
        end if
    end while
    return pop()
end function
```

Figure 3.12.: The Ackermann Function as a While Program

While-berechenbar

Definition 28 (While Computability) A function $f : \mathbb{N}^n \to_p \mathbb{N}$ is *while computable* if there exists a while program P such that for all $x_1, \ldots, x_n \in \mathbb{N}$ and memory $m : \mathbb{N} \to \mathbb{N}$ defined as

$$m(i) := \begin{cases} x_i & \text{if } 1 \leq i \leq n \\ 0 & \text{else} \end{cases}$$

the following holds:

- If $x_1, \ldots, x_n \in domain(f)$, then $[\![P]\!](m) : \mathbb{N} \to \mathbb{N}$ and $[\![P]\!](m)(0) = f(x_1, \ldots, x_n)$.
- If $x_1, \ldots, x_n \notin domain(f)$, then $[\![P]\!](m) = \bot$.

In other words, for a defined value of $f(x_1, \ldots, x_n)$, the program terminates with this value in variable x_0. If $f(x_1, \ldots, x_n)$ is undefined, the program does not terminate.

We will now show that while programs are indeed more powerful than loop programs by sketching how the previously presented Ackermann function, which is not loop computable, can be computed by a while program. In the left diagram of Figure 3.12 we show the recursive definition of the Ackermann function as a computer program with recursive function calls; in the right diagram the corresponding while program is shown (as previously discussed, we may make use of conditional statements and a loop with arbitrary conditions).

The core idea of the while program is to simulate the passing of arguments to function calls and the returning of results by the use of a *stack*, i.e., a sequence of values from which the value that was added last by a *push* operation is the one that is returned first by a *pop* operation. A recursive call of $ack(x_1, x_2)$ is simulated by pushing x_1 and x_2 (in the reverse order) on the stack; the program iteratively pops these values from the stack and then either replaces them by the function result (first branch) or by the arguments of a recursive call that will provide the result (second branch) or by the three arguments of the two nested calls (third branch): in the last case, the two last arguments for the inner call will be replaced by the result which, together with the first argument, will provide the argument for the outer call. If there is only one element left on the stack, this element denotes the overall function result.

Stapelspeicher

This example demonstrates how every program whose only recursive function calls are such that their results either denote the overall result or the argument of another recursive call can be translated to a while program; by storing additional information on the stack, even a translation of every kind of recursive function is possible (we omit the details).

Furthermore, it is interesting to note that, while the Ackermann function could not be computed by an arbitrary number of bounded loops, it could be computed by a *single* unbounded loop. This is actually an consequence of a very general result.

Theorem 15 (Kleene's Normal Form Theorem) Every while computable function can be computed by a while program of the following form (*Kleene's normal form*)

Kleenesche Normalform

```
x_c := 1;
while x_c do
  if x_c = 1 then P_1
  else if x_c = 2 then P_2
  ...
  else if x_c = n then P_n
  end if
end while
```

where the program parts $P_1, \ldots, P_n$ do not contain while loops.

The variable x_c in Kleene's normal form takes the role of a *control variable* that determines which of the n program parts P_i is to be executed next. The program starts with $x_c = 1$; when x_c becomes 0, the program terminates.

Goto-Programm

Proof We only sketch the proof whose core idea is to translate a while program into a *goto program*, i.e., a program which consists of a sequence of instructions of form L_i: P_i where L_i is a label that is unique for every instruction of the program; P_i can be either an assignment statement or a conditional jump statement **if** x_i **goto** L_j with the meaning that, if $x_i \neq 0$, then the next instruction to be executed is the one with label L_j, else it is the one with label L_{i+1} (see Section 3.2.4 for more details).

An unconditional jump statement **goto** L can be then simulated by x_j:=1; **if** x_j **goto** L. Furthermore, a conditional jump **if** $x_i = 0$ **goto** L_j can be simulated by the program

```
      y_i := 0
      if x_i goto L
      y_i := 1
L :   if y_i goto L_j
```

Now every while program

while x_i **do** P **end**

can be translated into a goto program

```
L_i :     if x_i = 0 goto L_{i+1}
          P;
          goto L_i
L_{i+1} : ...
```

Vice versa, every goto program

```
L_1 :  P_1
L_2 :  P_2
...
L_n :  P_n
```

can be translated into a while program in Kleene's normal form

```
x_c := 1;
while x_c do
  if x_c = 1 then x_c := 2; P_1
  else if x_c = 2 then x_c := 3; P_2
  ...
  else if x_c = n then x_c := 0; P_n
  end if
end while
```



```
(x_l, x_q, x_r) := input(x_0)
x_a := head(x_r)
while transition(x_q, x_a) do
  if x_q = q_1 ∧ x_a = a_1 then
    P_1
  else if x_q = q_2 ∧ x_a = a_2 then
    P_2
  else if ... then
    ...
  else if x_q = q_n ∧ x_a = a_n then
    P_n
  end
  x_a := head(x_r)
end
x_0 := output(x_l, x_q, x_r)
```

Figure 3.13.: Proof ⇒ of Theorem 16

where we replace every statement P_i of form **if** x_i **goto** L_j by **if** $x_i \neq 0$ **then** $x_c := j$ **end**.

Thus every while program can be first translated into a goto program, which can then be translated back into a while program in Kleene's normal form. □

Thus while programs are more powerful than loop programs, but we still have to discuss their relationship to Turing machines. This relationship is provided by the following result.

Theorem 16 (Turing Machines and While Programs) Every Turing machine can be simulated by a while program and vice versa.

Proof We sketch both directions of the proof.

⇒ The while program simulates the Turing machine with the help of three variables x_l, x_q and x_r that describe the current configuration: x_q represents the state, x_l represents the part of the tape left to the tape head, and x_r represents the part under/right to the head (excluding the infinite suffix ␣␣...); each of these variables holds an appropriate encoding of the corresponding part of the Turing machine configuration as a natural number.

Each of the n transition $\delta(q,a) = (q',a',d)$ of the Turing machine can be translated into a sequence of assignments on x_l, x_s, x_r; we thus get n while loop programs $P_1,\ldots,P_n$. The

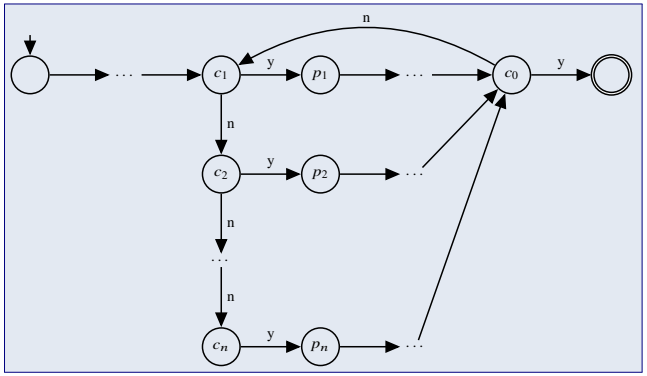


Figure 3.14.: Proof ⇐ of Theorem 16

overall program for the simulation of the Turing machine is then sketched in Figure 3.13. First it initializes from the variable x_0, which represents the initial state of the Turing machine tape, the variables x_l, x_q, x_r. Then it repeatedly reads the head symbol represented by x_r into the variable x_a and, based on x_q and x_a, checks whether there is some transition possible: if yes, P selects the appropriate transition i and performs the corresponding program P_i. If there is no more transition possible, the final content of the tape is copied back to x_0.

⇐ By Theorem 15, it suffices to demonstrate how a while program in Kleene's normal form can be simulated by a Turing machine. For a program with k variables, the machine uses k tapes each of which represents in some encoding the content of one program variable.

Each program part P_i is translated into a corresponding fragment of the Turing machine's transition function; this fragment starts with some state p_i and simulates the sequence of assignments performed by P_i via corresponding modifications of the tapes; afterwards it branches to some common state c_0 (see Figure 3.14). Likewise each test $x_c = i$ is translated into a transition function fragment that starts in some state c_i; if the test succeeds, it goes to the state p_i and to the state c_{i+1}, otherwise.

With these facilities at hand the Turing machine operates as follows: after simulating the assignment $x_c := 1$, it goes to state c_1 to start the simulation of the tests $x_c = 1, x_c = 2, \ldots$, until it has determined the number i such that $x_c = i$: it then goes to the state p_i which starts the simulation of P_i. When this simulation is finished, the machine is in state c_0 from where it starts the simulation of the test $x_c = 0$; if the test succeeds, the machine

terminates; otherwise it returns to state c_1.

The following theorem is a direct consequence of Theorem 16.

Theorem 17 (While Computability and Turing Computability) Let $f : \mathbb{N} \rightarrow_p \mathbb{N}$ be a partial function on natural numbers. Let $g : \{a\}^* \rightarrow_p \{a\}^*$ be a partial function on repetitions of some symbol a such that $domain(f) = domain(g)$ and $\forall n \in \mathbb{N} : g(a^n) = a^{f(n)}$, i.e., g maps repetitions of length n to repetitions of length $f(n)$. Then f is while computable if and only if g is Turing computable.

The theorem can be also generalized to n-ary functions; every while computable function is thus (for some suitable encoding of natural numbers) Turing computable and vice versa.

The computer science concepts of *loop* and *while* programs demonstrated in this section are closely related to mathematical concepts which we are now going to investigate.

3.2.3. Primitive Recursive and μ-recursive Functions

We turn our attention to a computational model that is not based on computer science concepts such as machines or programs but purely on mathematical functions over the natural numbers. These functions are defined by recursive equations of a certain form which we accept as "computable" in an intuitive sense.

Definition 29 (Primitive Recursion) The following functions over the natural numbers are called *primitive recursive*: primitiv recursiv

- **The constant null function** $0 \in \mathbb{N}$ is primitive recursive[2].
- **The successor function** $s : \mathbb{N} \rightarrow \mathbb{N}, s(x) := x + 1$ is primitive recursive.
- **The projection functions** $p_i^n : \mathbb{N}^n \rightarrow \mathbb{N}, p_i^n(x_1, \ldots, x_n) := x_i$ are primitive recursive.
- **Every function defined by composition**

$$h : \mathbb{N}^n \rightarrow \mathbb{N}$$
$$h(x_1, \ldots, x_n) := f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

  is primitive recursive, provided that $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is primitive recursive and $g_1, \ldots, g_k : \mathbb{N}^n \rightarrow \mathbb{N}$ are primitive recursive.

- **Every function defined by the primitive recursion scheme**

$$h : \mathbb{N}^{n+1} \to \mathbb{N}$$
$$h(y, x_1 \ldots x_n) := \begin{cases} f(x_1, \ldots, x_n) & \text{if } y = 0 \\ g(y-1, h(y-1, x_1, \ldots, x_n), x_1, \ldots, x_n) & \text{else} \end{cases}$$

  is primitive recursive, provided that the functions $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ are primitive recursive.

A mathematical function is therefore primitive recursive if it is one of the basic functions (constant null, successor, projection) or can be constructed from these functions by a finite number of applications of the principles of composition and/or primitive recursion. This primitive recursion principle can be better understood by unwinding the definition:

$$\begin{aligned}
h(0, x) &= f(x) \\
h(1, x) &= g(0, h(0, x), x) = g(0, f(x), x) \\
h(2, x) &= g(1, h(1, x), x) = g(1, g(0, f(x), x), x) \\
h(3, x) &= g(2, h(2, x), x) = g(2, g(1, g(0, f(x), x), x), x) \\
&\ldots \\
h(y, x) &= g(y-1, h(y-1, x), x) = g(y-1, g(y-2, \ldots, g(0, f(x), x), \ldots, x), x)
\end{aligned}$$

Thus, for $y > 0$, $h(y, x)$ describes the y-times application of g to a base value $f(x)$, i.e., the computation of $h(y, x)$ requires the evaluation of y recursive function applications.

Using "pattern matching", primitive recursion can be also formulated by two equations

$$\begin{aligned}
h(0, x_1 \ldots, x_n) &:= f(x_1, \ldots, x_n) \\
h(y+1, x_1 \ldots, x_n) &:= g(y, h(y, x_1, \ldots, x_n), x_1, \ldots, x_n)
\end{aligned}$$

or in a more computer-science oriented style as

$h(y, x_1 \ldots x_n) :=$
case y **of**
0 : $f(x_1, \ldots, x_n)$
$z+1$: $g(z, h(z, x_1, \ldots, x_n), x_1, \ldots, x_n)$

[2] Formally, $0 : () \to \mathbb{N}$ where () denotes the set whose only element is the zero-length tuple (); however, we simply write the constant 0 to denote the function application 0().

Example 17 (Primitive Recursive Functions) The addition $y + x$ on natural numbers x, y is primitive recursive, because we can give the following definition in "pattern matching" style:

$$\begin{aligned}
0 + x &:= x \\
(y+1) + x &:= (y + x) + 1
\end{aligned}$$

Using the fact that addition is primitive recursive, we can also show that multiplication $y \cdot x$ on natural numbers x, y is primitive recursive:

$$\begin{aligned}
0 \cdot x &:= 0 \\
(y+1) \cdot x &:= y \cdot x + x
\end{aligned}$$

Thus also exponentiation x^y on natural numbers x and y is primitive recursive:

$$\begin{aligned}
x^0 &:= 1 \\
x^{y+1} &:= x^y \cdot x
\end{aligned}$$

□

A function defined by primitive recursion is recursively evaluated a bounded number of times exactly as a loop in a loop program is iterated a bounded number of times. Thus it seems possible to convert a loop program into a primitive recursive function.

Example 18 Take the loop program

$x_0 := x_1$
$x_2 := 0$
loop x_1 **do**
  if $x_0 = x_1 \wedge p(x_2) = 1$ **then**
    $x_0 := x_2$
  end
  $x_2 := x_2 + 1$
end

where p is an arbitrary function. This program computes in x_0 the smallest natural number $n < x_1$ for which $p(n) = 1$ holds (respectively $x_0 = x_1$, if $p(n) \neq 1$ for all $n < x_1$). Assuming that for input $x_1 = 5$ we get output $x_0 = 3$, the following trace describes the values of the program variables before/after every iteration of the loop:

x_0	x_1	x_2
5	5	0
5	5	1
5	5	2
5	5	3
3	5	4
3	5	5

If p is primitive recursive, we can define a primitive recursive function $min(x_1) := loop(x_1, x_1)$ that computes the same result as the loop program above. This function uses the auxiliary function *loop* defined by the following primitive recursion scheme:

$$loop(x_2, x_1) := \begin{cases} x_1 & \text{if } x_2 = 0 \\ if(x_2 - 1, loop(x_2 - 1, x_1), x_1) & \text{else} \end{cases}$$

The function $loop(x_2, x_1)$ returns the value that is assigned to program variable x_0 for input x_1 by x_2 iterations of the **loop** in above program; the definition $min(x_1) := loop(x_1, x_1)$ indicates that we want to determine the value that is assigned to x_0 by x_1 iterations. The argument x_2 takes the role of the loop iteration counter in above program. If $x_2 = 0$, the loop is not executed at all and the result is the initial value x_1 of variable x_0.

If $x_2 > 0$, we compute by the recursive call $loop(x_2 - 1, x_1)$ the value of x_0 after $x_2 - 1$ iterations of the loop. From this, we determine the value of x_0 after one more iteration by application of the selection function

$$if(x_2, x_0, x_1) := \begin{cases} x_2 & \text{if } x_0 = x_1 \wedge p(x_2) = 1 \\ x_0 & \text{else} \end{cases}$$

This function determines the value that is assigned to the program variable x_0 by the **if** statement in the loop program above; *if* is primitive recursive because it can be shown that a function defined by case distinction is primitive recursive.

The recursive invocations of function *loop* arising from the evaluation of $min(5) = loop(5,5)$

$$\begin{aligned}
&loop(0,5) = 5 \\
&loop(1,5) = if(0, loop(0,5),5) = if(0,5,5) = 5 \\
&loop(2,5) = if(1, loop(1,5),5) = if(1,5,5) = 5 \\
&loop(3,5) = if(2, loop(2,5),5) = if(2,5,5) = 5 \\
&loop(4,5) = if(3, loop(3,5),5) = if(3,5,5) = 3
\end{aligned}$$

$$loop(5,5) = if(4, loop(4,5),5) = if(4,3,5) = 3$$

show that in the sequence of evaluations of $loop(x_2, x_1) = x_0$ the values (x_0, x_1, x_2) correspond to the trace of the program variables in above loop program. □

Indeed, the following theorem confirms that such a translation is always possible, even in both directions.

Theorem 18 (Primitive Recursion and Loop Computability) Every primitive recursive function is loop computable and vice versa.

Proof We sketch both directions of the proof.

⇒ Let h be a primitive recursive function. We show that h is loop computable by structural induction over the definition of m.

- If h is one of the basic functions, it is clearly loop computable.
- If h can be defined with the help of primitive recursive functions $f, g_1, \ldots, g_k$ as

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

then, by the induction hypothesis, we can assume that $f, g_1, \ldots, g_k$ are loop computable, i.e., there exist loop programs f, g_1, ..., g_k that compute these functions. We can then construct the loop program

```
y_1 := g_1(x_1,...,x_n);
y_2 := g_2(x_1,...,x_n);
...
y_k := g_k(x_1,...,x_n);
x_0 := f(y_1,...,y_k)
```

which computes $h(x_1, \ldots, x_n)$.

In above code, we denote by a function application of form

```
b := f(a_1,...,a_n);
```

the program that first backups the values of $x_1, \ldots, x_n$ and of all variables written by f into otherwise unused variables, then writes $a_1, \ldots, a_n$ into $x_1, \ldots, x_n$, executes f, copies the result from x_0 to b, and finally restores the original values of $x_1, \ldots, x_n$ and of all variables written by f from the backup. We omit the details.

– If h can be defined with the help of primitive recursive functions f, g as

$$h(y, x_1 \dots x_n) := \begin{cases} f(x_1, \dots, x_n) & \text{if } y = 0 \\ g(y-1, h(y, x_1, \dots, x_n), x_1, \dots, x_n) & \text{else} \end{cases}$$

then, by the induction hypothesis, we can assume that f, g are loop computable, i.e., there exist loop programs $\boxed{f}$ and $\boxed{g}$ that compute these functions. We can then construct the loop program

```
x_0 := f(x_1, ..., x_n);
x_y := 0;
loop y do
  x_0 := g(x_y, x_0, x_1, ..., x_n);
  x_y := x_y + 1
end
```

which computes $h(y, x_1, \dots, x_n)$ (the function applications in the code are interpreted as shown in the previous case).

This case is of particular interest: it shows that the "outside-in" computation of the value of the term $g(y-1, g(y-2, \dots, g(0, f(x))))$ by y primitive recursive calls can be replaced by the y-times iteration of a loop that computes the value "inside-out".

$\Leftarrow$ Let h be a loop computable function with k arguments whose loop program P uses variables $x_0, x_1, \dots, x_n$ (where $n \geq k$). Let $f_P : \mathbb{N}^{n+1} \to \mathbb{N}^{n+1}$ be the function that maps the initial values of these variables before the execution of P to their final values after the execution of P.

We show by induction on the definition of P that f_P is primitive recursive. If f_P is primitive recursive, then also h is primitive recursive, because we can define h by composition as

$$h(x_1, \dots, x_k) = var_0(f_P(0, x_1, \dots, x_k, 0, \dots, 0))$$

where $var_i(x_0, \dots, x_n) := x_i$.

One might note that a primitive recursive function must return a single natural number, while the result of f_P is actually a tuple of such numbers. However, this problem can be easily solved by encoding each tuple of natural numbers as a single number and by defining the usual tuple construction and selection operations as primitive recursive functions over numbers. To simplify our presentation, we do not make this encoding explicit but just assume that we can freely use this kind of "generalized" number functions.

- If P is of form $x_i := k$, then

$$f_P(x_0, \dots, x_n) := (x_0, \dots, x_{i-1}, k, x_{i+1}, \dots, x_n)$$

is primitive recursive, because tuple construction is primitive recursive.

- If P is of form $x_i := x_j \pm 1$, then

$$f_P(x_0, \dots, x_n) := (x_0, \dots, x_{i-1}, x_j \pm 1, x_{i+1}, \dots, x_n)$$

is primitive recursive, because tuple construction and basic arithmetic on natural numbers is primitive recursive.

- If P is of form $P_1; P_2$, then by the induction assumption, the functions f_{P_1} and f_{P_2} are primitive recursive. Then also the function

$$f_P(x_0, \dots, x_n) := f_{P_2}(f_{P_1}(x_0, \dots, x_n))$$

defined by composition of f_{P_1} of f_{P_1} is also primitive recursive.

- If P is of form **loop** x_i **do** P' **end**, then by the induction assumption, the function $f_{P'}$ is primitive recursive. Then also the function

$$f_P(x_0, \dots, x_n) := g(x_i, x_0, \dots, x_n)$$

is primitive recursive where g is defined by primitive recursion as

$$g(0, x_0, \dots, x_n) := (x_0, \dots, x_n)$$
$$g(m+1, x_0, \dots, x_n) := f_{P'}(g(m, x_0, \dots, x_n))$$

□

By Theorem 13, we know that several (even recursively defined) functions are not loop computable, notably the function $a \uparrow^n b$ and the Ackermann function. By Theorem 18, we thus also know that these functions are also not primitive recursive.

However, we have also seen that the model of while programs is more powerful than the one of loop programs, e.g., it can be used to compute the Ackermann function. We apparently still lack in our model of recursive functions the counter part of while loops which is needed to make the model Turing complete. The following definition provides this missing concept.

My-rekursiv

Definition 30 (μ-Recursion) A partial function over the natural numbers is *mu-recursive* (μ-recursive) if

- it is the constant null function, or the successor function, or a projection function, or
- can be constructed from other μ-recursive functions by composition or primitive recursion, or
- can be defined as a function $h : \mathbb{N}^n \rightarrow_p \mathbb{N}$ with

$$h(x_1, \ldots, x_n) := (\mu f)(x_1, \ldots, x_n)$$

where $f : \mathbb{N}^{n+1} \rightarrow_p \mathbb{N}$ is a μ-recursive function and $(\mu f) : \mathbb{N}^n \rightarrow_p \mathbb{N}$ is defined as

$$(\mu f)(x_1, \ldots, x_n) := \min \left\{ y \in \mathbb{N} \;\middle|\; \begin{array}{l} f(y, x_1, \ldots, x_n) = 0 \wedge \\ \forall z \leq y : (z, x_1, \ldots, x_n) \in domain(f) \end{array} \right\}$$

Thus $h(x_1, \ldots, x_n)$ denotes the smallest number y such that $f(y, x_1, \ldots, x_n) = 0$ and f is defined for all $z \leq y$; the result of h is undefined if no such y exists.

Please note that a μ-recursive function is in general partial because the principle of μ-recursion is the analogon to the unbounded while loop; like a while loop may yield a non-terminating program, the μ-operator may yield an undefined function result (think of (μf) as the function that starts a search for the smallest value y such that $f(y, x_1, \ldots, x_n) = 0$; this search may or may not terminate).

However, a μ-recursive function may be also total; in particular, since the definition of μ-recursion subsumes primitive recursion, every primitive recursive function is a total μ-recursive function. As we will see later, the converse is not true, i.e., by μ-recursion more functions (even total ones) can be defined than by primitive recursion. The inclusion relationship between the classes of primitive recursive functions, total μ-recursive functions, and partial μ-recursive functions is illustrated in Figure 3.15 (every inclusion is proper, i.e., the three classes are all really different).

Example 19 (μ-Recursive Function Definition) We consider the sequence of numbers de-

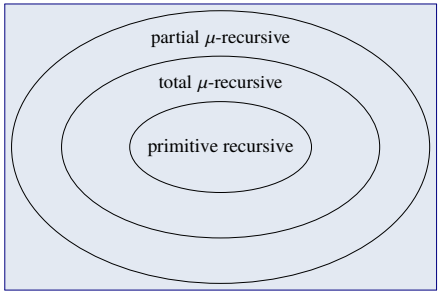


Figure 3.15.: Primitive Recursion and μ-Recursion

rived from the k-fold application

$$f^k(n) = \underbrace{f(f(f(\ldots f(n))))}_{k \text{ applications of } f}$$

of the function

$$f(n) := \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n + 1 & \text{otherwise} \end{cases}$$

to some start value $n \in \mathbb{N}$. For instance, if we take $n := 10$, we have

$$\begin{aligned} f^0(10) &= 10 \\ f^1(10) &= f(f^0(10)) = f(10) = 5 \\ f^2(10) &= f(f^1(10)) = f(5) = 16 \\ f^3(10) &= f(f^2(10)) = f(16) = 8 \\ f^4(10) &= f(f^3(10)) = f(8) = 4 \\ f^5(10) &= f(f^4(10)) = f(4) = 2 \\ f^6(10) &= f(f^5(10)) = f(2) = 1 \end{aligned}$$

Thus we have $f^6(10) = 1$. Interestingly, for every $n \in \mathbb{N}$, there seems to exist some $k \in \mathbb{N}$ such that $f^k(n) = 1$ (the so called "Collatz Conjecture"). We are now interested in defining the function $C(n)$ whose result is the minimal number k such that $f^k(n) = 1$; as above example demonstrates, we have $C(10) = 6$. If for some n no such k should exist (i.e., for starting value n

above process can be repeated forever without ultimately reaching 1), $C(n)$ is undefined.

The core idea of defining C as a μ-recursive function is as follows (the formally correct definition will be given below):

$$\begin{aligned} C(n) &:= (\mu D)(n) \\ D(k,n) &:= f^k(n) - 1 \\ f^k(n) &:= \begin{cases} n & \text{if } k = 0 \\ f(f^{k-1}(n)) & \text{otherwise} \end{cases} \end{aligned}$$

Here $f^k(n)$ is defined by primitive recursion where the parameter k determines the number of recursive applications. By the meaning of the μ-operator, $(\mu D)(n)$ denotes the smallest $k \in \mathbb{N}$ for which $D(k,n) = 0$; since $D(k,n) = 0$, if $f^k(n) = 1$, the function C is defined as intended.

Now the formally correct definition of C is as follows:

$$\begin{aligned} C(n) &:= (\mu D)(n) \\ D(k,n) &:= P(E(k,n)) \\ P(n) &:= \begin{cases} 0 & \text{if } n = 0 \\ p_1^2(n-1, P(n-1)) & \text{otherwise} \end{cases} \\ E(k,n) &:= \begin{cases} p_1^1(n) & \text{if } k = 0 \\ F(k-1, E(k-1,n), n) & \text{otherwise} \end{cases} \\ F(k,m,n) &:= f(p_2^3(k,m,n)) \end{aligned}$$

Here $P(n)$ denotes the predecessor $n-1$ of n (the result is 0, if $n = 0$) and $E(k,n)$ computes $f^k(n)$. Since thus P and E are indeed primitive recursive functions, the definition of C as a μ-recursive function is well-formed.

The Collatz Conjecture could be neither proved nor disproved yet; thus it is unknown whether C is total (and there may or may not exist also a primitive recursive definition of C).□

The close relationship between μ-recursive functions and while programs is stated by the following theorem.

Theorem 19 (μ-Recursion and While Computability) Every μ-recursive function is while computable and vice versa.

Proof We sketch both parts of the proof.

⇒ Let h be a μ-recursive function. We are going to show by induction on the definition of h the existence of a while program P that computes h.

If h is one of the basic functions or defined by composition or primitive recursion, the construction is as in the proof of Theorem 18. If h is defined by application of the μ-operator with the help of a μ-recursive function $f : \mathbb{N}^{n+1} \to_p \mathbb{N}$ as

$$\begin{aligned} &h : \mathbb{N}^n \to_p \mathbb{N} \\ &h(x_1, \ldots, x_n) := (\mu f)(x_1, \ldots, x_n) \end{aligned}$$

then there exists, by the induction hypothesis, a while program to compute

$$y := f(x_0, x_1, \ldots, x_n)$$

The program P to compute h is then as follows:

```
x_0 := 0;
y := f(x_0, x_1, ..., x_n);
while y do
    x_0 := x_0 + 1;
    y := f(x_0, x_1, ..., x_n)
end
```

When started in a state where the variables $x_1, \ldots, x_n$ hold the arguments of h, the variable x_0 holds after the execution of P the value of h.

⇐ Let $h : \mathbb{N}^k \to_p \mathbb{N}$ be a while computable function, i.e. there exists a while program P that uses $n + 1 > k$ variables $x_0, \ldots, x_n$ and computes h. We show that h is a μ-recursive function defined as

$$h(x_1, \ldots, x_k) := var_0(f_P(0, x_1, \ldots, x_k, 0, \ldots, 0))$$

where $var_i(x_0, \ldots, x_n) := x_i$ by proving the existence of a μ-recursive function $f_P : \mathbb{N}^{n+1} \to_p \mathbb{N}^{n+1}$ that maps the initial values of these variables before the execution of P to their final values after the execution of P; the proof proceeds by induction on the structure of P.

If P is an assignment, or a sequence, of a bounded loop, then the proof proceeds as in the proof of Theorem 18. If P is an unbounded loop

while x_i **do** P' **end**

then by the induction hypothesis there exists a μ-recursive function $f_{P'} : \mathbb{N}^n \to_p \mathbb{N}^n$ that maps the variable values before the execution of P to the values after the execution. Then f_P is defined as

$$f_P(x_0, \ldots, x_n) := g((\mu g_i)(x_0, \ldots, x_n), x_0, \ldots, x_n)$$

where

$$g_i : \mathbb{N}^{n+1} \to \mathbb{N}$$
$$g_i(m, x_0, \ldots, x_n) := var_i(g(m, x_0, \ldots, x_n))$$

and g is defined as in the proof of Theorem 18:

$$g(0, x_0, \ldots, x_n) := (x_0, \ldots, x_n)$$
$$g(m+1, x_0, \ldots, x_n) := f_{P'}(g(m, x_0, \ldots, x_n))$$

Thus $g(m, x_0, \ldots, x_n)$ denotes the values of the program variables after m iterations of P', $g_i(m, x_0, \ldots, x_n)$ denotes the value of variable i after m iterations, and $(\mu g_i)(x_1, \ldots, x_n)$ denotes the the minimum number of iterations required to drive variable x_i to 0. Then f_P is the value of the program variables after that number of iterations, i.e., when x_i for the first time becomes 0. □

Since the Ackermann function is while computable, it is by Theorem 19 a (total) μ-recursive function. Furthermore, we have sketched in the previous section, how every recursive function definition can be translated to a while program (by the use of a variable that simulates a stack of function calls). Thus every function that can be defined by some recursive definition can be also be defined in a μ-recursive form (by using a corresponding stack argument).

Since μ-recursive functions correspond to while programs, they also have a normal form.

Kleenesche Normalform

Theorem 20 (Kleene's Normal Form Theorem) Every μ-recursive function h can be defined in the following form (*Kleene's normal form*)

$$h(x_1, \ldots, x_k) := f_2(x_1, \ldots, x_k, (\mu g)(f_1(x_1, \ldots, x_k)))$$

where f_1, f_2, g are *primitive* recursive functions.

Proof Since h is μ-recursive, it is by Theorem 19 computable by a while program, which by Theorem 15 can be assumed to be in normal form:


```
x_c := 1;
while xc do ... end
```

As shown in the second parts of the proofs of Theorem 18 and Theorem 19, this program P can be translated into a function f_P

$$f_P(x_0, \ldots, x_n) := g((\mu g_c)(init(x_0, \ldots, x_n)), init(x_0, \ldots, x_n))$$

where $init(x_0, \ldots, x_c, \ldots, x_n) := (x_0, \ldots, 1, \ldots, x_n)$, and (since P does not contain any more while loops) g and g_c are primitive recursive. Then h is defined as

$$h(x_1, \ldots, x_k) := var_0(f_P(0, x_1, \ldots, x_k, 0, \ldots, 0))$$

i.e., by expanding the definition of f_P, as

$$h(x_1, \ldots, x_k) := var_0(g((\mu g_c)(init(0, x_1, \ldots, x_k, 0, \ldots, 0)), init(0, x_1, \ldots, x_k, 0, \ldots, 0)))$$

We can then define the primitive recursive functions

$$f_1(x_1, \ldots, x_k) := init(0, x_1, \ldots, x_k, 0, \ldots, 0)$$
$$f_2(x_1, \ldots, x_k, r) := var_0(g(r, init(0, x_1, \ldots, x_k, 0, \ldots, 0)))$$

and consequently have

$$h(x_1, \ldots, x_k) := f_2(x_1, \ldots, x_k, (\mu g_c)(f_1(x_1, \ldots, x_k)))$$ □

This section has demonstrated that the principle of recursion has the same computational power as loop iteration. In fact, all the results shown in the previous section for loop programs and while programs were historically derived *first* for primitive and μ-recursive functions (the theory of computability was originally called *recursion theory*); later these results were transferred to the looping constructs that arose in the emerging programming languages.

3.2.4. Further Models

Figure 3.16 summarizes our current picture of computational models and their relationships. In Chapter 2 we have presented the model of finite-state machines which we have shown to be less powerful than the model of Turing machines introduced in this chapter. We have shown in the last sections that there are several equally powerful models such as random access machines,

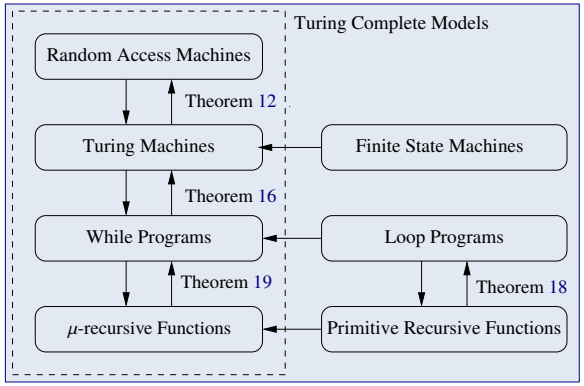


Figure 3.16.: Computational Models ($A \longrightarrow B$: A can be simulated by B)

while programs, and mu-recursives functions; furthermore we have identified the less powerful models of loop programs and primitive recursives functions.

To complete the picture, we will in the following shortly sketch various other computational models which are Turing complete (the list is not exhaustive, there are many more examples).

Goto Programs

We consider programs with jump instructions rather than loop constructs.

Goto-Programm

Definition 31 (Goto Programs) A *goto program* is a finite sequence of instructions

$$L_1 : P_1; L_2 : P_2; \ldots; P_n : A_n$$

where every action P_k is inductively defined by the following grammar:

$$P \ ::= \ x_i := 0 \mid x_i := x_j + 1 \mid x_i := x_j - 1 \mid \textbf{if } x_i \textbf{ goto } L_j$$

The meaning of a program P is defined by a partial function $[\![P]\!](k,m)$ which maps the initial state (k,m) of P, consisting of a program counter $k \in \mathbb{N}$ and memory $m : \mathbb{N} \to \mathbb{N}$, to its final state (unless the program does not terminate):

$$\begin{array}{lcl} [\![P]\!](0,m) & := & m \\ [\![P = \ldots; P_k : x_i := 0; \ldots]\!](k,m) & := & [\![P]\!](k+1, m[i \leftarrow 0]) \end{array}$$

$$\begin{array}{lcl} [\![P = \ldots; P_k : x_i := x_j + 1; \ldots]\!](k,m) & := & [\![P]\!](k+1, m[i \leftarrow m[j]+1]) \\ [\![P = \ldots; P_k : x_i := x_j - 1; \ldots]\!](k,m) & := & [\![P]\!](k+1, m[i \leftarrow \max\{0, m[j]-1\}]) \\ [\![P = \ldots; P_k : \textbf{if } x_i \textbf{ goto } L_j; \ldots]\!](\mathrm{k,m}) & := & \begin{cases} [\![P]\!](k+1,m), & \text{if } m(i) = 0 \\ [\![P]\!](j,m), & \text{if } m(i) \neq 0 \end{cases} \end{array}$$

The proof of Theorem 15 already sketched how while programs can be translated into goto programs (and vice versa), which shows that this model is indeed Turing complete.

Lambda Calculus

We now discuss the calculus which stood at the beginning of the *Church-Turing Thesis*.

Definition 32 (*Lambda calculus*) A *lambda term* (λ-term) T is defined by the following grammar: Lambda-Kalkül, Lambda-Term

$$T \ ::= x_i \mid (T\ T) \mid (\lambda x_i . T)$$

A term of form $(\lambda x_i . T)$ is called an *abstraction*, a term of form $(T\ T)$ is called an *application*. Abstraktion, Anwendung

The meaning of a λ-term is defined by the following reduction relation $\rightarrow$:

$$((\lambda x_i . T_1) T_2) \rightarrow (T_1[x_i \leftarrow T_2])$$

We denote by $T_1 \rightarrow^* T_2$ the existence of a sequence of reductions $T_1 \rightarrow \ldots \rightarrow T_2$. The final term T_2 is in *normal form*, if no further reduction is possible. Normalform

Intuitively, the abstraction term $T_1 = (\lambda x_i . T)$ with the binder operator λ represents a function with parameter x_i and result T; the application term $(T_1 T_2)$ applies this function to an argument T_2, which yields the result $T[x \leftarrow T_2]$, i.e. T where the parameter x has been replaced by argument T_2.

In general, in a given term T different reductions are possible yielding different reduction sequences $T \rightarrow \ldots$. However, as claimed by the following theorem (which holds also for certain other reduction systems), if a reduction sequence terminates in a normal form, then this form is uniquely defined.

Theorem 21 (Church-Rosser Theorem) If $T_1 \to^* T_2$ and $T_1 \to^* T_2'$ such that both T_2 and T_2' are in normal form, then $T_2 = T_2'$.

Now the claim is that every computable function can be represented by a λ-term such that the application of the term to an argument yields a reduction sequence that terminates in a normal form which represents the result of the function. Thus the core question is how the lambda calculus is able to mimic unbounded computations (as in while loops or μ-recursive functions).

Fixpunkt-Operatator

The answer lies in the fact that it is possible to define a λ-term Y (the *fixed point operator*) such that, for any lambda term F, we have the reduction sequence

$$(YF) \to^* (F(YF))$$

i.e. YF represents a *fixed point* of F. With the help of this operator, we can in the λ-calculus mimic any recursive function definition, e.g.,

$$f(x) := \ldots f(g(x)) \ldots$$

by defining the lambda terms

$$F := \lambda h.\lambda x. \ldots h(g(x)) \ldots$$
$$f := YF$$

We then have

$$fa = (YF)a \to^* F(YF)a \to^* \ldots (YF)(g(a)) \ldots = \ldots f(g(a)) \ldots$$

i.e., the reduction yields the desired recursive application of f; the lambda calculus is thus indeed Turing complete.

It remains to give a suitable definition of of the fixed point operator:

$$Y := (\lambda f.((\lambda x.(f(xx)))(\lambda x.(f(xx)))))$$

We leave it to the reader to check that this definition indeed yields $(YF) \to^* (F(YF))$.

The lambda calculus is the formal basis of functional programming languages which consider computations not as updates of programs states (as in imperative languages) but as constructions

of result values.

Rewriting Systems

While lambda calculus reduces its terms by a fixed set of rules, we may consider systems that reduce terms by a user defined rule set.

Definition 33 (Term Rewriting System) A *term rewriting system* is a set of rules of form

Termersetzungssystem

$$L \to R$$

where L and R are terms such that L is not a variable and every variable that appears in R must also appear in L. A term T can be rewritten to another term T', written as $T \to T'$, if there is some rule $L \to R$ and a substitution σ (a mapping of variables to terms) such that

- some subterm U of T matches the left hand side L of the rule under the substitution σ, i.e., $U = L\sigma$,
- T' is derived from T by replacing U with $R\sigma$, i.e with the right hand side of the rule after applying the variable replacement.

We denote by $T_1 \to^* T_2$ is a sequence of rewritings $T_1 \to \ldots \to T_2$. The final term T_2 is in *normal form*, if no further reduction is possible.

Normalform

An example of a term rewriting system is

$$f(x, f(y, z)) \to f(f(x, y), z)$$
$$f(x, e) \to x$$
$$f(x, i(x)) \to e$$

from which we have, for instance, the reduction sequences

$$f(a, f(i(a), e)) \to f(f(a, i(a)), e) \to f(e, e) = e$$
$$f(a, f(i(a), e)) \to f(a, i(a)) \to e$$

It is not very hard to see that by term rewriting systems we can encode arbitrary recursive function definitions and that thus the model is Turing complete.

The set of objects to be reduced need not necessarily be terms; there are e.g. also string rewriting systems (the objects are strings of symbols) or graph rewriting systems (the objects are graphs). Rewriting systems have many practical applications, they play a role in automated theorem proving as well as in software engineering (many tools for the *model driven architecture* approach are based on graph rewriting).

3.3. The Chomsky Hierarchy

In Chapter 2, we have introduced the concept of finite-state machines whose languages (the regular languages) are those that can be also described by regular expressions. In this chapter, we have introduced Turing machines that recognize the recursively enumerable languages; for these languages, we are now going to introduce a corresponding alternative description based on the concept of formal grammars.

Grammatik

Definition 34 (Grammar) A *grammar* G is a quadruple (N, Σ, P, S) consisting of

Nonterminalsymbol
- a finite set N of elements which we call *nonterminal symbols*,

- a finite set Σ disjoint from N, i.e.,

  $$N \cap \Sigma = \emptyset$$

  Terminalsymbol
  whose elements which we call *terminal symbols*,

Produktionsregel
- a finite set P of *production rules* of form $l \to r$ such that

  $$l \in (N \cup \Sigma)^* \circ N \circ (N \cup \Sigma)^*$$
  $$r \in (N \cup \Sigma)^*$$

  i.e., the left side l and the right side r consist of sequences of nonterminal and/or terminal symbols where l must contain at least one nonterminal symbol,

Startsymbol
- a nonterminal symbols $S \in N$ which we call the *start symbol*.

If a grammar has multiple production rules $l \to r_1, l \to r_2, \ldots, l \to r_n$ we may abbreviate these to a single rule

$$l \to r_1 \mid r_2 \mid \ldots \mid r_n$$

Definition 35 (Language of a Grammar) Let $G = (N, \Sigma, P, S)$ be a grammar and $w_1, w_2 \in (N \cup \Sigma)^*$ two words of nonterminal and/or terminal symbols.

direkte Ableitung
- There exists a *direct derivation* $w_1 \Rightarrow w_2$ in grammar G, if there are words $u, v \in (N \cup \Sigma)^*$ of nonterminal and/or terminal symbols such that

  $$w_1 = ulv$$
  $$w_2 = urv$$

  and $l \to r$ is a production rule of G.

Ableitung
- There exists a *derivation* $w_1 \Rightarrow^* w_2$ in G if there exists a sequence of zero or more direct derivations $w_1 \Rightarrow \ldots \Rightarrow w_2$ in G.

Satzform
- A word $w \in (N \cup \Sigma)^*$ of nonterminal and/or terminal symbols is a *sentential form* in G, if there is a derivation $S \Rightarrow^* w$ in G starting with the start symbol S.

Satz
- A sentential form w is a *sentence* if w contains only terminal symbols, i.e. $w \in \Sigma^*$.

- The language $L(G)$ of grammar $G = (N, \Sigma, P, S)$ is the set of all its sentences:

  $$L(G) := \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Example 20 (Finite Grammar Language) Take the grammar $G = (N, \Sigma, P, S)$ with

$$N = \{S, A, B\}$$
$$\Sigma = \{a, b, c\}$$
$$P = \{S \to Ac,\ A \to aB, A \to BBb, B \to b, B \to ab\}$$

We then have e.g. the two derivations

$$S \Rightarrow Ac \Rightarrow aBc \Rightarrow abc$$
$$S \Rightarrow Ac \Rightarrow BBbc \Rightarrow abBbc \Rightarrow ababbc$$

The language of the grammar is

$$L(G) = \{abc, aabc, bbbc, babbc, abbbc, ababbc\}$$

i.e. it consists of finitely many words. □

Example 21 (Infinite Grammar Language) Take the grammar $G = (N, \Sigma, P, S)$ with

$$N = \{S\}$$
$$\Sigma = \{\text{'('}, \text{')'}, \text{'['}, \text{']'}\}$$
$$P = \{S \to \varepsilon \mid SS \mid [S] \mid (S)\}$$

This grammar has a *recursive* production rule where the nonterminal symbol S on the left also occurs on the right which gives rise to infinitely many possible derivations, e.g.

$$S \Rightarrow [S] \Rightarrow [SS] \Rightarrow [(S)S] \Rightarrow [()S] \Rightarrow [()[S]] \Rightarrow [()[(S)]] \Rightarrow [()[()]]$$

Thus the language of the grammar (the so-called *Dyck-Language*) is infinite; it consists of all words that have matching pairs of parentheses "()" and brackets "[]". □

We can now represent regular languages by certain kinds of grammars.

rechtslinear

Definition 36 (Right Linear Grammars) A grammar $G = (N, \Sigma, P, S)$ is *right linear* if each rule in P is of one of the forms

- $A \to \varepsilon$
- $A \to a$
- $A \to aB$

where $A, B \in N$ are nonterminal symbols and $a \in \Sigma$ is a terminal symbol.

Theorem 22 (Regular Languages and Right-Linear Grammars) The languages of right linear grammars are exactly the regular languages, i.e., for every right linear grammar G, there exists a finite-state machine M with $L(M) = L(G)$ and vice versa.

Proof We sketch both directions of the proof.

⇒ We construct from the right linear grammar G a NFSM M. The states of M are the nonterminal symbols of G extended by an additional accepting state q_f. The start state of M is the start symbol of G. The construction of the automaton proceeds according to the production rules of G:

- For every rule $A \to \varepsilon$, the state A becomes accepting.
- For every rule $A \to a$, we add a transition $\delta(A, a) = q_f$.
- For every rule $A \to aB$, we add a transition $\delta(A, a) = B$.

⇐ Let M be a DFSM. We take as the set of nonterminal symbols of G the set of states of M and as the start symbol the start state of M.

- For every transition $\delta(q, a) = q'$ we add a production rule $q \to aq'$.
- For every accepting state q, we add a production rule $q \to \varepsilon$. □

If we drop any restriction on the form of the grammar, we get a corresponding equivalence result for recursively enumerable languages.

Theorem 23 (Recursively Enumerable Languages and Unrestricted Grammars) The languages of (unrestricted) grammars are exactly the recursively enumerable languages, i.e., for every grammar G, there exists a Turing machine M with $L(M) = L(G)$ and vice versa.

Proof We sketch both parts of the proof.

⇒ We construct from the grammar G a 2-tape nondeterministic Turing machine M. This machine uses the second tape to construct some sentence of $L(G)$: it starts by writing S on the tape and then nondeterministically chooses some production rule $l \to r$ of G and applies it to some occurrence of l on the tape, replacing this occurrence by r. Then M checks whether the result equals the word on the first tape. If yes, M accepts the word, otherwise, it continues with another production rule. If the word can be derived from G, there is some run of M which detects this derivation, thus $L(M) = L(G)$.

⇐ We can construct from the Turing machine M a grammar G whose sentential forms encode pairs (w, c) of the input word w and the current configuration c of M; every such sentential form contains a non-terminal symbol such that by application of a rule, the sentential form representing the pair of input word and current machine configuration is replaced by a sentential form of input word and successor configuration. The production rules of G are defined such that

- from the start symbol of G, every pair (w, c) of an input word w and a corresponding start configuration c of M can be derived;

- for every transition of M a corresponding production rule is constructed, such that if the transition moves configuration c to c', in G a corresponding derivation $(w, c) \Rightarrow (w, c')$ is possible;
- if in M from a configuration c no further transition is possible and the corresponding machine state is accepting, then in G the derivation $(w, c) \Rightarrow w$ is possible (if the state is not accepting, from (w, c) no derivation is possible).

The sentences of G are thus the words accepted by M, thus $L(G) = L(M)$. □

By the second part of Theorem 23, (unrestricted) grammars represent another Turing complete computational model.

We have thus constructed a correspondence between two levels of machine models (finite-state machines and Turing machines), their languages (regular languages and recursively enumerable languages) and grammars generating these languages (right linear grammars and unrestricted grammars). These levels are the two ends of a hierarchy of grammars, languages, and machine models, which was conceived in 1956 by the linguist Noam Chomsky.

Chomsky-Hierarchie

Theorem 24 (*Chomsky hierarchy*) For type $i \in \{0, 1, 2, 3\}$, let $G(i)$, $L(i)$, and $M(i)$ be the following sets of grammars, languages and machine models:

Type i	Grammar $G(i)$	Language $L(i)$	Machine $M(i)$
0	unrestricted	recursively enumerable	Turing machine
1	context-sensitive	context-sensitive	linear bounded automaton
2	context-free	context-free	pushdown automaton
3	right linear	regular	finite-state machine

We say that a grammar/language/machine is of type i if it is in set $G(i)/L(i)/M(i)$.

Then, for every type i, we have:

- $L(i)$ is the set of languages of grammars $G(i)$ and machines $M(i)$.
- For $i > 0$, the set of languages of type $L(i)$ is a proper subset of the set of languages $L(i-1)$, i.e. $L(i) \subset L(i-1)$.
- For $i > 0$, every machine in $M(i)$ can be simulated by a machine in $M(i-1)$ (but not vice versa).

We are now going to complete the missing layers of this hierarchy.

Definition 37 (Type 2 Grammars and Machines) A grammar G is *context-free* if every rule has form $A \to r$ where $A \in N$ is a nonterminal symbol. kontextfrei

A *pushdown automaton* M is a nondeterministic finite-state machine that has an unbounded stack of symbols as a *working memory*: in every transition $\delta(q, a, b) = (q', w)$, M reads the next input symbol a (a may be ε, i.e., M may also not read a symbol) and the symbol b on the top of stack, and replaces b by a (possibly empty) sequence w of symbols. Kellerautomat

The name "context-free" stems from the fact that, given a rule $A \to r$, in a sentential form any occurrence of A may be replaced by r, independent of the context (the surrounding terminal or nonterminal symbols) in which it occurs.

Example 22 (Type 2 Language) The language

$$L := \{a^i b^i \mid i \in \mathbb{N}\}$$

is of type 2 (but, as we have already proved by application of the Pumping Lemma, not of type 3), because it is the language of the following grammar:

$$S \to \varepsilon \mid aSb$$

We can e.g. construct the derivation

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$ □

Context-free languages play a very important role in computer science, e.g., most programming languages are of this form; compiler generators take context-free grammars as input and produce as output parsers, i.e., compiler front ends that read and syntax check source code (such parsers are in essence implementations of pushdown automata).

Definition 38 (Type 1 Grammars and Machines) A grammar G is *context-sensitive* if kontextsensitiv

- in every rule $l \to r$, we have $|l| \leq |r|$, i.e., the length of left side l is less than or equal the length of right side r,
- the rule $S \to \varepsilon$ is only allowed if the start symbol S does not appear on the right hand side of any rule.

Linear beschränkter Automat

A *linear bounded automaton* M is a nondeterministic Turing machine with k tapes such that, for an input word of length n, the machine never moves the tape head beyond the first n cells of each tape.

Linear beschränkter Automat

In contrast to a general Turing machine, a *linear bounded automaton* must therefore cope with a limited amount of space (a fixed multiple of the length of the input word).

Example 23 (Type 1 Language) The language

$$L := \{a^i b^i f^i \mid i \in \mathbb{N}\}$$

is of type 1 (but not of type 2, as can be shown by a generalized Pumping Lemma for context-free languages), because it is the language of the following grammar:

$$S \to \varepsilon \mid T, T \to ABC \mid TABC$$
$$BA \to AB, CB \to BC, CA \to AC$$
$$AB \to ab, bC \to bc, Aa \to aa, bB \to bb, cC \to cc$$

Every nonterminal A, B, C represents one occurrence of a, b, c in the final word. The first two rules generate an arbitrary number of triples A, B, C. The next three rules sort A, B, C such that they occur in the right order. The last five rules replace those occurrences of A, B, C that occur in the right order by a, b, c.

We can then e.g. construct the derivation

$$\underline{S} \Rightarrow \underline{T} \Rightarrow \underline{T}ABC \Rightarrow ABC\underline{A}BC \Rightarrow A\underline{BA}CBC \Rightarrow AAB\underline{CB}C \Rightarrow A\underline{AB}BCC$$
$$\Rightarrow \underline{Aa}bBCC \Rightarrow aa\underline{bB}CC \Rightarrow aabb\underline{CC} \Rightarrow aabbc\underline{C} \Rightarrow aabbcc \qquad \square$$

It is not hard to see that the language $\{a^i b^j f^k \mid k = ack(i, j)\}$ where ack denotes the Ackermann function is a type 0 language; one can also show that it is not a type 1 language. We thus have now examples of languages for each type of the Chomsky hierarchy:

- **Type 3:** $\{(ab)^n \mid n \in \mathbb{N}\}$
- **Type 2:** $\{a^n b^n \mid n \in \mathbb{N}\}$
- **Type 1:** $\{a^n b^n f^n \mid n \in \mathbb{N}\}$
- **Type 0:** $\{a^i b^j f^k \mid k = ack(i, j)\}$

None of these languages of type i is also of type $i + 1$.

3.4. Real Computers

We have in this chapter presented Turing machines and various equivalent computational models and proved their superiority over finite-state machines. But what is the core reason of this superiority? In a nutshell, while a finite-state machine has only a bounded number of states, the other models support computations with *arbitrarily many states*:

- in a Turing machine, these states are represented by the content of the tape (which is infinitely long and may thus hold symbol sequences of arbitrary length);
- in a random access machine, these states are represented by the content of the memory (which has infinite size and may thus hold arbitrarily many natural numbers each of which may be even of arbitrary size);
- in a while program respectively μ-recursive function, the states are represented by variables that may hold natural numbers of arbitrary size and may thus encode arbitrarily large data structures.

Furthermore, while in finite-state machines the length of a computation is a priori bounded by the size of the input, computations in these models may run for an arbitrary amount of time and even not terminate at all (this is the price that a model has to pay to become Turing complete).

One may now ask what kind of model actually captures the computational power of a *real computer*. Unfortunately, the answer is not so simple:

- One may argue that a real computer is made of various components such as processor, memory, and communication devices, which are in turn composed of finitely many digital elements. Since the state of a computer in every clock cycle is determined by the current Boolean values of all these elements, it can be only in finitely many different states and is thus only a finite-state machine. In particular, a real computer cannot simulate the infinite tape of a Turing machine, i.e., it is *not* Turing complete. It can also not implement natural numbers of arbitrary size and thus not unbounded arithmetic.

  This more hardware-oriented view of a computer is taken by *model checkers*, programs that verify whether a finite state (hardware/software) system satisfies certain correctness properties by investigating all possible executions of the system.

- However, one may also argue that a computer has in principle access to an arbitrary amount of memory (e.g. by the technique of *virtual memory* which is backed by an arbitrary amount of disk space). Allowing arbitrarily much memory, a computer *is* Turing

complete, in particular, it can simulate the arbitrarily large initial part of the tape that is written by a Turing machine. We can in a computer also represent arbitrarily big natural numbers by sequences of memory words and thus implement unbounded natural number arithmetic (this is exactly what computer algebra systems do).

This more idealized view of a computer is taken by algorithm theory, which considers that, always when a concrete computer runs out of memory during a computation, we can add *on demand* more memory to let the computation run through.

Whether we consider a real computer as a finite-state machine or as a Turing complete computational model, is a matter of the point of view respectively of the goal of the consideration.

Chapter 4.

Limits of Computability

In Chapter 3, we have investigated the power of Turing machines and other Turing complete computational models. In this chapter, we will explore their limits by showing that there exist computational problems that are *not* solvable in these models. We restrict our consideration mainly to *decision problems*, i.e., problems that have "yes/no" answers and define what it means to *decide* such problems. We then discuss the "mother of all undecidable problems", the famous *halting problem*. Subsequently, we show that also other problems are undecidable (by reducing the halting problem to these problems); finally, we can even prove that *all* non-trivial questions about the behavior of Turing machines are undecidable.

4.1. Decision Problems

In this chapter, we focus on the following kind of computational problems.

Definition 39 (Decision Problem) Let Σ be an alphabet. A *decision problem* (short *problem*) $P \subseteq \Sigma^*$ over Σ^* is a property of words over Σ (i.e., a subset of Σ^*). We write $P(w)$ to indicate that word $w \in \Sigma^*$ has property P, i.e., that $w \in P$.

Entscheidungsproblem
(Entscheidungs)problem

Typically, a decision problem P is defined by a formula

$$P(w) :\Leftrightarrow \ldots \text{ (formula with free variable } w)$$

which is interpreted as defining the set

$$P := \{w \in \Sigma^* \mid \ldots\}$$

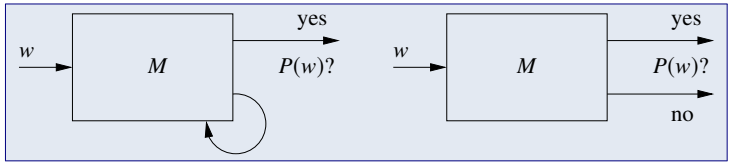


Figure 4.1.: Semi-Decidability versus Decidability

If we read $P(w)$ as the decision question

> Does word w have property P?

the set P thus consists of all words w for which the answer to the question is "yes".

Example 24 (Decision Problem) Let $\Sigma = \{0\}$ and the decision problem be

> Is the length of w a square number?

Then we can formally define the problem as

$$P(w) :\Leftrightarrow \exists n \in \mathbb{N} : |w| = n^2$$

or, equivalently, as

$$P := \{w \in \Sigma^* \mid \exists n \in \mathbb{N} : |w| = n^2\}$$

In other words, P consists of all the words whose length is a square number:

$$P := \{\varepsilon, 0, 0000, 000000000, \ldots\}$$ □

The following definition relates problems to the languages of Turing machines.

Definition 40 (Semi-Decidability and Decidability)

- A problem P is *semi-decidable*, if P is a recursively enumerable language. — semi-entscheidbar
- A problem P is *decidable*, if P is a recursive language. — entscheidbar

For a semi-decidable problem P, there exists a Turing machine M which accepts a word w if and only if $P(w)$ holds (see Figure 4.1). If $P(w)$ does not hold, then M may not terminate; thus

we are only sure to receive an answer to the decision question $P(w)$ in a finite amount of time only, if the answer is positive; we may wait forever for an answer if this answer were negative. In contrast, for a decidable problem P, there exists a Turing machine M which always halts and which accepts w if and only if $P(w)$ holds. Thus we are sure to receive an answer to the question $P(w)$ in a finite amount of time, be it a positive or a negative one.

We can translate our previously established results about the languages of Turing machines to the decidability of problems.

Theorem 25 (Decidabilty of Complement) If a problem P is decidable, also its complement $\overline{P}$ is decidable.

The complement $\overline{P}$ is derived from P by negating the question respectively by swapping the answer: a "yes" to P becomes a "no" to $\overline{P}$; a "no" to P becomes a "yes" to $\overline{P}$.

Proof This theorem is an immediate consequence of Definition 40 and Theorem 9. □

Theorem 26 (Decidability) A problem P is decidable if and only if both P and its complement $\overline{P}$ are semi-decidable.

Proof This theorem is an immediate consequence of Definition 40 and Theorem 8. □

We can also characterize decidability in terms of the computability of functions.

Theorem 27 (Decidability and Computability)

- A problem $P \subseteq \Sigma^*$ is semi-decidable if and only if the *partial characteristic function* $1'_P : \Sigma^* \rightarrow_p \{1\}$ with $domain(1'_P) = P$ is Turing computable, where — partielle charakteristische Funktion

$$1'_P(w) := \begin{cases} 1 & \text{if } P(w) \\ \text{undefined} & \text{if } \neg P(w) \end{cases}$$

charakteristische Funktion

- A problem $P \subseteq \Sigma^*$ is decidable if and only if the *characteristic function* $1_P : \Sigma^* \to \{0,1\}$ is Turing computable, where:

$$1_P(w) := \begin{cases} 1 & \text{if } P(w) \\ 0 & \text{if } \neg P(w) \end{cases}$$

Proof We sketch both parts of the proof.

- If P is semi-decidable, there exists a Turing machine M such that, for every word $w \in P = domain(1'_P)$, M accepts w. We can then construct a Turing machine M' which takes w from the tape and calls M on w. If M accepts w, M' writes 1 on the tape.

  Vice versa, if $1'_P$ is Turing computable, there exists a Turing machine M such that, for every word $w \in P = domain(1'_P)$, M accepts w and writes 1 on the tape. We can then construct a Turing machine M' which takes w from the tape and calls M on w. If M writes 1, M' accepts w.

- The proof of this part proceeds like the proof of the first part, except that M is always terminating. If M does not accept w, then M' writes 0; respectively if M writes 0, then M' does not accept w. □

We will now turn our attention to problems that are not decidable.

4.2. The Halting Problem

The problems which we are mainly going to investigate are decision problems about Turing machines. Since decision problems have to be expressed over words in some alphabet, we need a possibility to encode a Turing machine as a word.

Code einer Turing-Maschine

Theorem 28 (Turing Machine Code) Let TM be the set of all Turing machines. Then there exists a function $\langle . \rangle : TM \to \{0,1\}^*$ from Turing machines to bit strings, where we call the bit string $\langle M \rangle \in \{0,1\}^*$ the *Turing machine code* of Turing machine M, such that

1. different Turing machines have different codes, i.e., if $M \neq M'$, then $\langle M \rangle \neq \langle M' \rangle$;
2. we can recognize valid Turing-machine codes, i.e., the problem $w \in range(\langle . \rangle)$ is decidable.

Proof The core idea is to assign to all machine states, alphabet symbols, and tape directions unique natural numbers and to encode every transition $\delta(q_i, a_j) = (q_k, a_l, d_r)$ by the tuple of numbers (i, j, k, l, r) in binary form; we omit the details. □

A Turing machine code is sometimes also called a *Gödel number* because Kurt Gödel defined a similar encoding of logic statements for the proof of his *incompleteness theorem*. Using Turing machine codes, we can formulate the central problem of this section.

Halteproblem

Definition 41 (Halting Problem) The *halting problem* HP is to decide, for given Turing machine code $\langle M \rangle$ and word w, whether M halts on input w:

$$HP := \{(\langle M \rangle, w) \mid \text{Turing machine } M \text{ halts on input word } w\}$$

Here the pair (w_1, w_2) denotes a word that appropriately encodes both w_1 and w_2 such that a Turing machine can extract from the encoding both w_1 and w_2 again.

While this problem is apparently of central interest in computer science, we have the following negative result.

Theorem 29 (Undecidability of Halting Problem) The halting problem is undecidable.

For the proof of this theorem, we need some further concepts.

Theorem 30 (Word Enumeration) Let Σ be an alphabet. Then there exists an enumeration $w = (w_0, w_1, \ldots)$ of all words over Σ, i.e. for every word $w' \in \Sigma^*$, there exists an index $i \in \mathbb{N}$ such that $w' = w_i$.

Proof The enumeration w starts with the empty word, then lists the all words of length 1 (of which there are finitely many), then lists all the words of length 2 (of which there are finitely many), and so on. By construction every word eventually appears in w. □

Theorem 31 (Turing Machine Enumeration) There is an enumeration $M = (M_0, M_1, \ldots)$ of all Turing machines, i.e., for every Turing machine M' there exists an index $i \in \mathbb{N}$ such that $M' = M_i$.

Proof Let $C = (C_0, C_1, \ldots)$ be the enumeration of all Turing machine codes in bit-alphabetic word order. We define M_i as the Turing machine denoted by C_i (which is unique, since different Turing machines have different codes). Since every Turing machine has a code and C enumerates all codes, M is the enumeration of all Turing machines. □

With the help of these results, we can prove our core theorem.

Proof (Undecidability of Halting Problem) We define the function $h : \mathbb{N} \times \mathbb{N} \to \{0,1\}$ such that

$$h(i,j) := \begin{cases} 1 & \text{if Turing machine } M_i \text{ halts on input word } w_j \\ 0 & \text{otherwise} \end{cases}$$

If the halting problem were decidable, then h were computable by a Turing machine: given a Turing machine M that decides the halting problem, we could construct a Turing machine M_h that computes h as follows: M_h takes its input pair (i,j) and computes $\langle M_i \rangle$ and w_j (by enumerating the Turing machine codes $\langle M_0 \rangle, \ldots, \langle M_i \rangle$ and words $w_0, \ldots, w_j$). It then passes the pair $(\langle M_i \rangle, w_j)$ to M which eventually halts. If M accepts its input, then M_h returns 1, else it returns 0.

It thus suffices to show that h is not computable by a Turing machine. For this purpose, we assume that h is computable and derive a contradiction.

First, we define as function $d : \mathbb{N} \to \{0,1\}$ as

$$d(i) := h(i,i)$$

i.e., $d(i) = 1$, if and only if Turing machine M_i terminates on input word w_i. The values $d(0) = h(0,0), d(1) = h(1,1), d(2) = h(2,2), \ldots$ are the entries of the diagonal of the table of


```
function M(w):
    let i ∈ ℕ such that w = w_i
    case d(i) of
        0: return yes
        1: loop end loop
    end case
end function
```

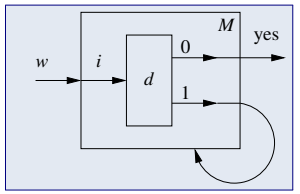


Figure 4.2.: Proof that the Halting Problem is Undecidable

the values of function h

h	$j=0$	$j=1$	$j=2$	$\ldots$
$i=0$	$\mathbf{h(0,0)}$	$h(0,1)$	$h(0,2)$	$\ldots$
$i=1$	$h(1,0)$	$\mathbf{h(1,1)}$	$h(1,2)$	$\ldots$
$i=2$	$h(2,0)$	$h(2,1)$	$\mathbf{h(2,2)}$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

our proof is therefore based on a so called *diagonalization* argument. Since h is computable, also d is computable.

Next, we define the Turing machine M depicted in Figure 4.2: M takes a word w and determines its index $i \in \mathbb{N}$ in the enumeration of all words, i.e., $w = w_i$. Then M computes $d(i)$: if the result is 0, M terminates; if the result is 1, M does not terminate.

Now let i be the index of M in the enumeration of all Turing machines (i.e., $M = M_i$), and give w_i as input to M. By construction, M for input w_i halts if and only if $d(i) = 0$. Since $d(i) = h(i,i)$, by the definition of h, we have $d(i) = 0$, if and only if Turing machine M_i does not halt for input w_i. Thus M halts for input w_i if and only if M_i does not halt for input w_i. But since M itself is M_i, this represents a contradiction. □

The core "trick" in above proof is that it is possible, by passing to M a word w_i with $M_i = M$, to let M draw some conclusion abouts its own behavior. Such a form of "self-reference" leads to many classical paradoxes (the liar paradox: "A Cretan says: 'all Cretans lie'.") and is also at the core of the proof of *Gödel's incompleteness theorem* which states that in any sufficiently rich formal system there are true statements that are not provable in that system.

```
function M(w):
    w' ← f(w)
    return M'(w')
end function
```

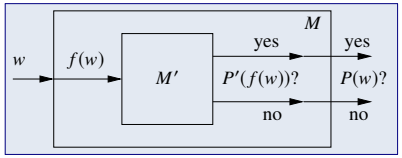


Figure 4.3.: Reduction Proof

4.3. Reduction Proofs

In the following we are going to show that many other interesting problems in computer science are not decidable. The core proof strategy is expressed by the following definition and accompanying theorem.

reduzierbar

Definition 42 (Reducibility) A decision problem $P \subseteq \Sigma^*$ is *reducible* to a decision problem $P' \subseteq \Gamma^*$ (written $P \leq P'$) if there is a computable function $f : \Sigma^* \to \Gamma^*$ such that

$$P(w) \Leftrightarrow P'(f(w))$$

In other words, w has property P if and only if $f(w)$ has property P'.

Theorem 32 (Reduction Proof) For all decision problems P and P' with $P \leq P'$, it holds that, if P is not decidable, then also P' is not decidable.

Proof It suffices to show that, if P' is decidable, then P is decidable. Thus we assume that P' is decidable and show that P is decidable. Since P' is decidable, there is a Turing machine M' that decides P'. We are going to construct a Turing machine M that decides P (see Figure 4.3).

Since $P \leq P'$, there is a computable function f such that $w \in P \Leftrightarrow f(w) \in P'$. Since f is computable, machine M can take its input w and compute $f(w)$ which it passes to M'. M' accepts $f(w)$ if and only if $P'(f(w))$ holds. M accepts w if and only if M' accepts $f(w)$. □

To show that a problem P' is not decidable, it thus suffices to show that a problem which has been previously proved to be not decidable (e.g. the halting problem) is reducible to P'. For example, we may show that also a restricted form of the halting problem is undecidable.


```
function M_H(⟨M⟩, w):
    ⟨M'⟩ := compute(⟨M⟩, w)
    return M_R(⟨M'⟩)
end function
```

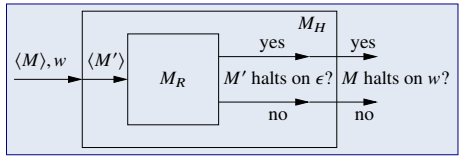


Figure 4.4.: Proof that the Restricted Halting Problem is Undecidable

Theorem 33 (Undecidability of Restricted Halting Problem) Let the *restricted halting problem RHP* be to decide, for given Turing machine M, whether M halts for input ε:

eingeschränktes Halteproblem

$$RHP := \{\langle M \rangle \mid \text{Turing machine } M \text{ halts on input word } \varepsilon\}$$

The restricted halting problem is not decidable.

Proof We prove this theorem by showing that the halting problem is reducible to the restricted halting problem. Thus we assume that the restricted halting problem is decidable and show that then also the halting problem is decidable.

Since the restricted halting problem is decidable, there exists a Turing machine M_R such that M_R accepts input c, if and only if c is the code of a Turing machine M which halts on input ε. We can then define the following Turing machine M_H, which accepts input (c, w), if and only if c is the code of a Turing machine M that terminates on input w (see Figure 4.4): if c is not a well-formed Turing machine code, then M_H does not accept its input. Otherwise, M_H constructs from (c, w) the code c' of the Turing machine M' which first prints w on its tape and then behaves like M. In other words, M' terminates for input ε (which is ignored and overwritten by w) if and only if M terminates on input w.

Then M_H passes c' to M_R:

- If M_R accepts c', then M' halts on input ε, and thus M halts on input w. In this case, M_H accepts (c, w).
- If M_R does not accept c', then M' does not halt on input ε and thus M does not halt on input w. In this case, M_H does not accept (c, w).

Since M_H thus decides the halting problem, we have a contradiction to Theorem 29. □

```
function M_H(⟨M⟩, w):
    ⟨M'⟩ := compute(⟨M⟩)
    return M_A(⟨M'⟩, w)
end function
```

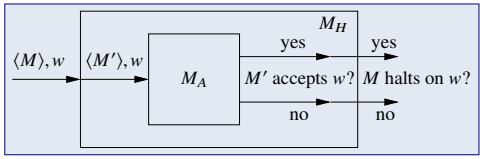


Figure 4.5.: Proof that the Acceptance Problem is Undecidable

Since also the restricted halting problem is apparently reducible to the general halting problem, both problems are in a certain sense "equally difficult".

In the same style also other problems can be shown to be undecidable.

Akzeptanz-Problem

Theorem 34 (Undecidability of Acceptance Problem) Let the *acceptance problem* AP be to decide, for given Turing machine M and word w, whether M accepts w:

$$AP := \{(\langle M \rangle, w) \mid w \in L(M)\}$$

The acceptance problem is not decidable.

Proof We prove this theorem by showing that the halting problem is reducible to the acceptance problem. Thus we assume that the acceptance problem is decidable and show that then also the halting problem is decidable.

Since the acceptance problem is decidable, there exists a Turing machine M_A such that M_A accepts input (c, w), if and only if c is the code of a Turing machine M which accepts w.

We can then define the following Turing machine M_H, which accepts input (c, w), if and only if c is the code of a Turing machine M that halts on input w (see Figure 4.5). If c is not a well-formed Turing machine code, then M_H does not accept its input. Otherwise, M_H slightly modifies c to the code c' of the Turing machine M' which behaves as M, except that in those cases where M halts and does not accept its input w, M' halts and accepts w. In other words, M' accepts its input w, if and only if M halts on input w.

Then M_H passes (c', w) to M_A:

- If M_A accepts (c', w), then M' accepts w, and thus M halts. In this case, M_H accepts (c, w).
- If M_A does not accept (c, w), then M' does not accept w and thus M does not halt. In this case, M_H does not accept (c, w).

Since M_H thus decides the halting problem, we have a contradiction to Theorem 29. □

While we may have negative results about the decidability of problems, we may still have positive results about their semi-decidability.

Theorem 35 (Semi-Decidability of the Acceptance Problem) The acceptance problem is semi-decidable.

Proof The core idea of the proof is the construction (which we do not show in detail) of the *universal Turing machine* M_u whose language is AP (the language of the acceptance problem is correspondingly also called the *universal language*). universelle Turing-Maschine universelle Sprache

This machine M_u is "universal" in the sense that it is an *interpreter* for Turing machine codes: given input (c, w), M_u simulates the execution of the Turing machine M denoted by c for input w:

- If the real execution of M halts for input w with/without acceptance, then also the simulated execution halts with/without acceptance; thus M_u accepts its input (c, w) if in the simulation M has accepted w.
- If the real execution of M does not halt for input w, then also the simulated execution does not halt; thus M_u does not accept its input (c, w).

Thus M_u semi-decides AP. □

With the help of the universal Turing machine, it is also possible to reduce the acceptance problem to the halting problem: assume that there exists a Turing machine M_H which decides the halting problem. Then we can construct the following Turing machine M_A which decides the acceptance problem: from input (c, w), M_A constructs a particular machine M_{cw} and invokes M_H with input $(\langle M_{cw} \rangle, \varepsilon)$ such that M_H accepts this input if and only if the Turing machine with code c accepts input w.

Since M_H decides the halting problem, it suffices to construct M_{cw} such that it halts on input ε if and only if the Turing machine with code c accepts input w. To achieve this, M_{cw} ignores its own input and invokes the universal Turing machine M_u with input (c, w); if M_u halts and accepts this input, then also M_{cw} halts and accepts its input. If M_u does not accept its input (because it does not halt or because it halts in a non-accepting state), then M_{cw} does not halt. Thus M_{cw} halts if and only if M_u accepts input (c, w) which is the case if and only if the Turing machine with code c accepts input w.

Since we have also shown in the proof of Theorem 34 that the halting problem can be reduced to the acceptance problem, both problems are in a certain sense "equivalent". One may also use the diagonalization argument presented in the previous section to show directly that the acceptance problem is not decidable; then it follows from above reduction that the halting problem is not decidable.

From the semi-decidability of the acceptance problem, we can now derive our first problem which is not even semi-decidable.

Theorem 36 (The Non-Acceptance Problem) The non-acceptance problem defined as

$$NAP := \{(\langle M\rangle, w) \mid w \notin L(M)\}$$

is not semi-decidable.

Proof We note that the non-acceptance problem is in essence the complement of the acceptance problem, i.e.

$$NAP = \overline{AP'}$$

where $AP' = AP \cup ERR$ and ERR is the set of all pairs (c, w) where c does not represent a valid Turing machine code.

Since by Theorem 28, ERR is decidable and by Theorem 35, AP is semi-decidable, also AP' is semi-decidable. However, by Theorem 34, AP is not decidable, and thus also AP' is not decidable.

Since thus AP' is semi-decidable but not decidable, by Theorem 26, its complement NAP is not even semi-decidable. □

We can derive corresponding results for the halting problem.

Theorem 37 (Semi-Decidability of the Halting Problem)

- The halting problem is semi-decidable.
- The non-halting problem defined as

  $$NHP := \{(\langle M\rangle, w) \mid \text{Turing machine } M \text{ does not halt for input word } w\}$$

  is not semi-decidable.

Proof The first part can be shown by reducing the halting problem to the acceptance problem in analogy to the proof of Theorem 34; from the semi-decision of the acceptance of w by M', we can derive a corresponding semi-decision of the termination of M on input w.

The second part of the theorem follows from the first part and Theorem 26 in the same way as in the proof of Theorem 36. □

We thus have established the following relationships

Problem	semi-decidable	decidable
Halting	yes	no
Non-Halting	no	no
Acceptance	yes	no
Non-Acceptance	no	no

and also shown that the halting problem and the acceptance problem are equally difficult, in the sense that each problem can be reduced to the other one.

4.4. Rice's Theorem

In the previous section, we have shown that several interesting properties of the behavior of a Turing machine (Does it halt? Does it accepts its input?) are not decidable (but only semi-decidable). Actually, these are only special cases of a general class of properties.

Definition 43 (Property of Recursively Enumerable Language) Let S be a set of recursively enumerable languages (i.e., a property of recursively enumerable languages).

- We call S *non-trivial* if there is at least one recursively enumerable language that is in S and there is at least one recursively enumerable language that is not in S (i.e, some recursively enumerable languages have the property and some do not have it). nicht-trivial
- We call S *decidable* if the problem entscheidbar

  $$P_S := \{\langle M\rangle \mid L(M) \in S\}$$

  is decidable, i.e., if it is decidable, whether the language of a given Turing machine M has property S.

Since a recursively enumerable language is the language of a Turing machine, it characterizes the input/output behavior of a Turing machine. A property of a recursively enumerable language should therefore be essentially considered as the property of the input/output behavior of a Turing machine.

In 1951, Henry Rice was able to prove the following theorem which generalizes the previously discussed undecidability results in that it severely limits what properties we can decide about Turing machines.

Satz von Rice

Theorem 38 (*Rice's Theorem*) Every non-trivial property of recursively enumerable languages is undecidable.

In other words, there is no Turing machine which for every possible Turing machine M can decide, whether the language of M has a non-trivial property; thus no non-trivial question about the input/output behavior of Turing machines (or other Turing complete computational models) is decidable. Consequently, also no nontrivial property of computable functions is decidable.

But beware: Rice's Theorem does not rule out that for *some* given Turing machine a decision about a non-trivial property of its input/output behavior is possible; e.g., we may prove for some (indeed infinitely many) Turing machines that they terminate for every possible input. Nevertheless, it is not possible to devise a general method that allows to perform such a decision for *all* possible Turing machines. Furthermore, the theorem does not apply to those properties of a Turing machine that are not covered by its language respectively input/output behavior (*non-functional properties*), e.g., how much time the machine takes or how much space it consumes for producing its output.

Proof Let S be a non-trivial property of recursively enumerable languages. Without loss of generality, we may assume that S does not contain the empty language $\emptyset$ (otherwise, we prove that the complement $\overline{S}$ which does not contain $\emptyset$ is not decidable, from which it follows by Theorem 25, that also S is not decidable). We assume that S is decidable and derive a contradiction.

Since S is decidable, there is a Turing machine M_S which decides S. Let us assume that it is possible, for every pair $(\langle M\rangle, w)$ of the code $\langle M\rangle$ of a Turing machine M and and word w, to compute the code $\langle M'\rangle$ of a Turing machine M' such that

$$L(M') \in S \Leftrightarrow w \in L(M)$$


```
function M_A(⟨M⟩, w):
    ⟨M'⟩ := compute(⟨M⟩)
    return M_S(⟨M'⟩)
end function
```

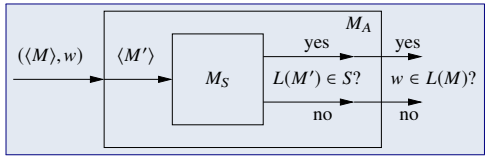


Figure 4.6.: Proof of Rice's Theorem (Part 1)

```
function M'(x):
    if w ∈ L(M) then
        if x ∈ L(M_L) then
            return yes
        else
            loop end loop
        end if
    else
        loop end loop
    end if
end function
```

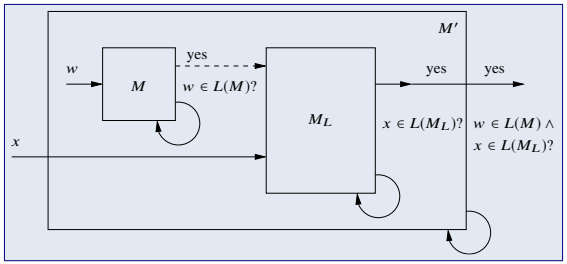


Figure 4.7.: Proof of Rice's Theorem (Part 2)

i.e, that M' has property S if and only if M accepts w (we will show below that the computation of $\langle M'\rangle$ is indeed possible).

Then we can construct a Turing machine M_A which decides the acceptance problem (see Figure 4.6): M_A takes its input (c, w). If c is not a valid Turing machine code, then M_A does not accept its input. Otherwise, c is the code $\langle M\rangle$ of some Turing machine M. Then M_A computes from $(\langle M\rangle, w)$ the Turing machine code $\langle M'\rangle$ which it passes to M_S:

- If M_S accepts $\langle M'\rangle$, then $L(M') \in S$, therefore by construction of M', $w \in L(M)$. In this case, M_A accepts its input.
- If M_S does not accept $\langle M'\rangle$, then $L(M') \notin S$, therefore by construction of M', $w \notin L(M)$. In this case, M_A does not accepts its input.

Thus M_A decides the acceptance problem, which is a contradiction to Theorem 34.

It remains to show, how M_A can compute from $(\langle M\rangle, w)$ the Turing machine code $\langle M'\rangle$.

First, M_A enumerates all possible Turing machine codes and applies M_S to decide whether this code has property S. Since S is non-trivial, it is not empty; therefore M_A will eventually detect the code $\langle M_L\rangle \in S$ of some Turing machine M_L with $L(M_L) \in S$.

Then M_A constructs the code $\langle M'\rangle$ of a Turing machine M' which operates as follows (see Figure 4.7): M' ignores for the moment its input word x and applies the universal Turing

machine M_u to $(\langle M\rangle, w)$, i.e., it simulates the execution of M on input w:

- If the simulated execution of M does not accept its input w (e.g., by not halting), then also M' does not accept its input x (e.g., by not halting).
- If the simulated execution of M accepts its input w, then M' applies M_u to simulate the execution of M_L on its input x. if M_L accepts x, then also M' accepts x; if M_L does not accept x (e.g., by not halting), then also M' does not accept x (e.g., by not halting).

Consequently, M' accepts its input x only if M accepts w and if M_L accepts x. In other words, if M does not accept w, M' does not accept any input; if M accepts w, then M' accepts the same words as M_L; the language $L(M')$ of M' therefore is

$$L(M') = \begin{cases} \emptyset & \text{if } w \notin L(M) \\ L(M_L) & \text{if } w \in L(M) \end{cases}$$

We know $\emptyset \notin S$ and $L(M_L) \in S$. Thus $L(M') \in S \Leftrightarrow w \in L(M)$. □

By Rice's Theorem, many interesting problems about Turing machines are undecidable:

- The halting problem (also in its restricted form).
- The acceptance problem $w \in L(M)$ (also in its restricted form $\varepsilon \in L(M)$).
- Leerheitsproblem — The *emptiness problem*: is $L(M)$ empty?
- Endlichkeit einer Sprache — The problem of *language finiteness*: is $L(M)$ finite?
- Sprachäquivalenz — The problem of *language equivalence*: $L(M_1) = L(M_2)$?
- Spracheinschluss — The problem of *language inclusion*: $L(M_1) \subseteq L(M_2)$?
- The problem whether $L(M)$ is regular, context-free, context-sensitive.

By Theorem 25 also the complements of these problems are not decidable; however, some of these problems (respectively their complements) may be semi-decidable.

It may be also shown that a problem completely unrelated to Turing machines is undecidable, typically by reducing an undecidable problem about Turing machines to this problem:

- Entscheidungsproblem — The *Entscheidungsproblem*: Given a formula and a finite set of axioms, all in first order predicate logic, decide, whether the formula is valid in every structure that satisfies the axioms.

This problem (formulated by David Hilbert in 1928) stood at the beginning of computability theory. It was the trigger for Church and Turing's work; they were (independently) able to show in 1936 and 1937 that it is undecidable (it is semi-decidable, i.e., the set of all valid formulas is enumerable; its complement is not semi-decidable, i.e., the set of all non-valid formulas is not enumerable).

- Post'sches Korrespondenzproblem — *Post's Correspondence Problem*: given a sequence of pairs $(x_1, y_1), \ldots, (x_n, y_n)$ of non-empty words x_i and y_i, find a sequence $i_1, \ldots, i_k$ of indices such that

$$x_{i_1} \ldots x_{i_k} = y_{i_1} \ldots y_{i_k}$$

- Wortproblem — The *word problem* for groups: given a group with finitely many generators $g_1, \ldots, g_n$, find two sequences of indices $i_1, \ldots, i_k$, $j_1, \ldots, j_l$ such that

$$g_{i_1} \circ \ldots \circ g_{i_k} = g_{j_1} \circ \ldots \circ g_{j_l}$$

- Zweideutigkeitsproblem — The *ambiguity problem* for context-free grammars: are there two different derivations for the same sentence?

These problems are of practical interest; the theory of decidability/undecidability has thus a profound impact on many areas in computer science, mathematics, and logic.

Part II.

Complexity

Chapter 5.

Basics of Complexity

We now turn our attention from *what* can be computed, to *how efficiently* it can be computed. We start by general considerations on the analysis of the *complexity* of computations, then investigate various notions of *asymptotic* complexity and their manipulation, and finally illustrate, how these notions give rise to various complexity *classes*.

5.1. Complexity of Computations

We are interested in investigating the *complexity*, i.e, the resource consumption, of computations. The main resources that we consider are

- time and
- space (memory).

Rather than investigating the complexity for each individual input separately, we consider the complexity of classes of *comparable* inputs, namely inputs of the same *size*. For each such class we are interested in determining

- the maximum complexity for all inputs of the class, and
- the average complexity for these inputs.

With the help of Turing machines, these intuitive notions can be given a precise meaning.

Definition 44 (Complexity) Let M be a Turing machine with input alphabet Σ that halts for every input. Let $I = \Sigma^*$ be the set of input words and $|i| \in \mathbb{N}$ be the size of input word $i \in I$.

We define the time consumption $t : I \to \mathbb{N}$ of M such that $t(i)$ denotes the number of moves that M makes for input i until it halts. Correspondingly, we define the space consumption $s : I \to \mathbb{N}$ such that $s(i)$ denotes the largest distance from the beginning of the tape that the tape head of M reaches for input i until M halts.

Then we define the *worst-case time complexity* (short *time complexity*) $T : \mathbb{N} \to \mathbb{N}$ and the *worst-case space complexity* (short *space complexity*) $S : \mathbb{N} \to \mathbb{N}$

Zeitkomplexität im schlechtesten Fall
Zeitkomplexität
Raumkomplexität im schlechtesten Fall
Raumkomplexität

$$T(n) := \max\{t(i) \mid i \in I \wedge |i| = n\}$$
$$S(n) := \max\{s(i) \mid i \in I \wedge |i| = n\}$$

as the maximum time/space that M consumes for any input of size n.

Furthermore, let *Input* be a family of (discrete) random variables that describe the distribution of inputs of each size n in I, i.e., $Input_n$ has a probability function $p_I^n : I \to [0,1]$ such that $p_I^n(i)$ denotes the probability that, among all inputs of size n, input i occurs.

Then we define the *average-case time complexity* $\overline{T} : \mathbb{N} \to \mathbb{N}$ and the *average-case space complexity* $\overline{S} : \mathbb{N} \to \mathbb{N}$

durchschnittliche Zeitkomplexität
durchschnittliche Raumkomplexität

$$\overline{T}(n) := E[Time_n]$$
$$\overline{S}(n) := E[Space_n]$$

as the expected values of the random variables $Time_n$ and $Space_n$ with probability functions $p_T^n : \mathbb{N} \to [0,1]$ and $p_S^n : \mathbb{N} \to [0,1]$ such that $p_T^n(t)$ respectively $p_S^n(s)$ denote the probabilities that the execution of M with an input of size n consumes time t respectively space s (assuming that inputs of size n from I are distributed according to $Input_n$).

While we have defined the complexity measures in terms of Turing machines, these notions can be generalized to any computational model by defining the input size $|i|$ and the time and space consumption $t(i)$ and $s(i)$ for that model. Apparently, different computational models thus may yield different complexities for the "same algorithm"; we will investigate in Chapter 7 in more detail the relationship among the complexities of different models.

As the following example shows, the analysis of worst-case complexity is generally much simpler than the analysis of average-case complexity: while, for given n, in worst-case complexity it is only one case that has to be considered, it is in average-case complexity a whole set of cases, such that reasoning about the probabilities of these cases is required.

Example 25 (Find the Maximum) Given a non-empty integer array a, we want to find the minimum index j such that $a[j] = \max\{a[i] \mid 0 \le i < length(a)\}$. The problem is apparently

solved by the following algorithm in pseudo-code:

```
j := 0; m := a[j]; i := 1                    1
while i < length(a)                          n
  if a[i] > m then                           n − 1
    j := i; m := a[j]                        N
  i := i + 1                                 n − 1
```

For analyzing the performance of this algorithm, we define as the input size $|a|$ the number $length(a)$ of elements in a, as $t(a)$ the number of times that each line of the algorithm is executed, and as $s(a)$ the number of variables used (including the number of elements of a).

Since the algorithm uses three variables i, j, k in addtion to a, its space complexity is

$$S(n) = \overline{S}(n) = n + 3$$

As for the time complexity, we indicate by the numbers shown to the right of the algorithm above, how often each line is executed for input size n. The only varying quantity is the number of times N that the line $j := i; m := a[j]$ is executed.

It is clear that in the worst case $N := n - 1$, therefore the worst-case time complexity is

$$T(n) = 1 + n + (n-1) + (n-1) + (n-1) = 4n - 2$$

For the average case analysis, let us for simplicity assume that a holds n distinct values. Since N depends only on the relative order of elements in a, we may identify these with the values $\{1, \ldots, n\}$. If we assume that all permutations are equally probable, we have

$$p_I^n(i) := \frac{1}{n!}$$

since there are $n!$ permutations of n values. Thus N becomes a random variable and our goal is to determine the expected value $E[N]$ of N.

Suppose we can determine the probability p_{nk} that $N = k$ for an array of size n. Then we have, since $0 \le N \le n - 1$,

$$p_{n0} + p_{n1} + p_{n2} + \ldots + p_{n,n-1} = \sum_{k=0}^{n-1} p_{nk} = 1$$

Since $p_{nk} = 0$ for $k \ge n$, we can also write the infinite sum

$$p_{n0} + p_{n1} + p_{n2} + \ldots = \sum_k p_{nk} = 1$$

Then we can compute the expected value of N as the sum of all products of the probability of $N = k$ and value k, i.e.,

$$E[N] = p_{n0} \cdot 0 + p_{n1} \cdot 1 + p_{n2} \cdot 2 + \ldots = \sum_k p_{nk} \cdot k$$

Thus our goal is to determine the value of this sum. To do so, we define the function

$$G_n(z) := p_{n0} \cdot z^0 + p_{n1} \cdot z^1 + p_{n2} \cdot z^2 + \ldots = \sum_k p_{nk} \cdot z^k$$

which is called the *generating function* of the infinite sequence $p_{n0}, p_{n1}, \ldots$. We compute the derivative of this function

$$G_n'(z) = p_{n0} \cdot 0 \cdot z^{-1} + p_{n1} \cdot 1 \cdot z^0 + p_{n2} \cdot 2 \cdot z^1 + \ldots = \sum_k p_{nk} \cdot k \cdot z^{k-1}$$

and consequently have

$$G_n'(1) = p_{n0} \cdot 0 + p_{n1} \cdot 1 + p_{n2} \cdot 2 + \ldots = \sum_k p_{nk} \cdot k$$

Since thus $E[N] = G_n'(1)$, our goal is to determine $G_n'(1)$.

For $n = 1$, we know $p_{10} = 1$ and $p_{1k} = 0$ for all $k \ge 1$, therefore

$$G_1'(1) = 1 \cdot 0 + 0 \cdot 1 + 0 \cdot 2 + \ldots = 0$$

For $n > 1$, we know that, if the loop has already found the maximum of the first $n - 1$ array elements, the last iteration of the loop will either increment N (if the last element is the largest among the n elements) or it will leave N as it is (if the last element is not the largest one). In the first case (which has probability $1/n$), the value of N becomes k, only if the value of N is $k - 1$ for the first $n - 1$ elements. In the second case (which has probability $(n-1)/n$), the value of N becomes k, only if the value of N is k for the first $n - 1$ elements. We thus can describe the

combined probability that the value of N is k for n array elements as

$$p_{nk} = \frac{1}{n} \cdot p_{n-1,k-1} + \frac{n-1}{n} \cdot p_{n-1,k}$$

We can determine from the definition of G the corresponding relationship for G as

$$G_n(z) = \frac{1}{n} \cdot z \cdot G_{n-1}(z) + \frac{n-1}{n} \cdot G_{n-1}(z)$$

By simplification, we get

$$G_n(z) = \frac{z+n-1}{n} \cdot G_{n-1}(z)$$

From this we can compute the derivation

$$G'_n(z) = \frac{1}{n} \cdot G_{n-1}(z) + \frac{z+n-1}{n} \cdot G'_{n-1}(z)$$

We have, for all n,

$$G_n(1) = p_{n0} + p_{n1} + p_{n2} + \ldots = \sum_k p_{nk} = 1$$

and can therefore derive

$$G'_n(1) = \frac{1}{n} \cdot 1 + \frac{1+n-1}{n} \cdot G'_{n-1}(1)$$

which we can simplify to

$$G'_n(1) = \frac{1}{n} + G'_{n-1}(1)$$

Summarizing, we have derived the recurrence relation (see Section 6.2)

$$G'_1(1) = 0$$
$$G'_n(1) = \frac{1}{n} + G'_{n-1}(1), \text{ if } n > 1$$

which can be easily solved as

$$G'_n(1) = \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \sum_{k=2}^{n} \frac{1}{k} = H_n - 1$$

where $H(n) = \sum_{k=1}^{n} \frac{1}{k}$ denotes the n-th harmonic number. For this number, there exists the well-known approximation $H(n) = \ln n + \gamma + \varepsilon_n$ where $\gamma \approx 0.577$ is *Euler's constant* and ε_n is a positive value less than $1/(2n)$. We have therefore derived

$$E[N] = \ln n + \gamma + \varepsilon_n - 1$$

The algorithm thus has the average-case time complexity

$$\overline{T}(n) = 1 + n + (n-1) + E[N] + (n-1) = 3n + \ln n + \varepsilon_n + \gamma - 2$$ □

Generally, in complexity analysis we are not really interested in all the details of the complexity functions, but we would like to capture their "overall behavior", especially for large inputs. For example, rather than stating

$$\overline{T}(n) = 3n + \ln n + \varepsilon_n + \gamma - 2$$

we might just say "$\overline{T}(n)$ is of the order $3n + \ln n$" or, since $3n$ is much larger than $\ln n$, even "$\overline{T}(n)$ is of the order $3n$". Ultimately we may even just focus on the growth of the time complexity as being in proportion with the input size by stating "$\overline{T}(n)$ is linear" which is typically expressed as

$$\overline{T}(n) = O(n)$$

In the following, we will discuss the formal underpinning of such approximations.

5.2. Asymptotic Complexity

In this section, we are going to introduce some basic notions that allow us to express the *growth patterns* of complexity functions in a succinct way.

O-Notation
Omega-Notation
Theta-Notation

Definition 45 (*Big-O notation*, *Big-Ω notation*, *Big-Θ notation*) Let $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function from $\mathbb{N}$ into the set $\mathbb{R}_{\geq 0}$ of the non-negative real numbers.

- $O(g)$ denotes the set of all functions $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

$$\exists c \in \mathbb{R}_{>0}, N \in \mathbb{N} : \forall n \geq N : f(n) \leq c \cdot g(n)$$

We write $f(n) = O(g(n))$ (read "$f(n)$ is (big) oh of $g(n)$") to indicate $f \in O(g)$; we also say that "f is bounded from above by g".

- $\Omega(g)$ denotes the set of all functions $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

$$\exists c \in \mathbb{R}_{>0}, N \in \mathbb{N} : \forall n \geq N : f(n) \geq c \cdot g(n)$$

  We write $f(n) = \Omega(g(n))$ (read "$f(n)$ is (big) omega of $g(n)$") to indicate $f \in \Omega(g)$; we also say that "f is bounded from below by g".

- $\Theta(g)$ denotes the set of all functions $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

$$f \in O(g) \wedge f \in \Omega(g)$$

  We write $f(n) = \Theta(g(n))$ (read "$f(n)$ is (big) theta of $g(n)$") to indicate $f \in \Theta(g)$; we also say that "f is bounded from above and below by g".

Landau-Symbol

The symbols O, Ω, Θ are called *Landau symbols*, after the mathematician Edmund Landau who popularized their use. Among these notions, the most widespread one is $f(n) = O(g(n))$; it is often also used when actually $f(n) = \Theta(g(n))$ is meant (i.e., also when $g(n)$ is not only a bound from above but also a bound from below).

One should also note that the statement $f(n) = O(g(n))$ clearly represents an abuse of notation, since it does not denote equality of two entities (f is a function, $O(g)$ is a set) but the containment of the first entity in the second, i.e., it should be better written $f(n) \in O(g(n))$. However, this notation has arisen from the practice of reading "=" as "is" (rather than "equals"); since it has become universally adopted, we will also stick to it.

Furthermore, in $f(n) = O(g(n))$, $f(n)$ and $g(n)$ are actually not functions but terms with a single free variable (which need not be "n"). To derive the intended interpretation $f \in O(g)$, we must first determine the free variable and from this the corresponding function definitions. This may become ambiguous if the terms also involve other symbols that actually denote constants. We must then deduce from the context the variable and the constants.

Example 26 The statement

> Let $c > 1$. Then $x^c = O(c^x)$.

should be interpreted as

> Let $c > 1$, $f(x) := x^c$, and $g(x) := c^x$. Then $f \in O(g)$. □

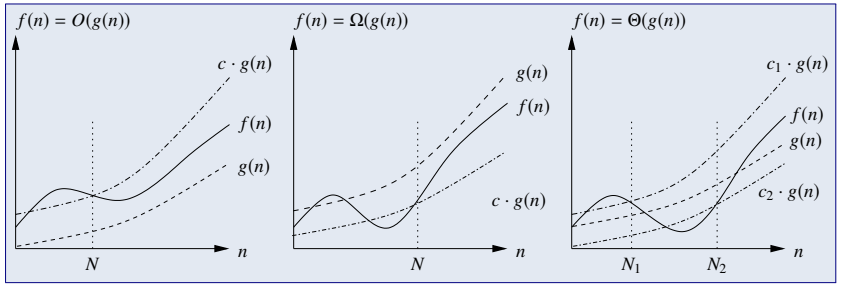


Figure 5.1.: The Landau Symbols

Intuitively, $f(n) = O(g(n))$ expresses the fact that function f "does not grow faster" than function g, i.e., that f is "bounded from above" by g. However:

- The bound need not hold for all arguments; it suffices if it holds from a certain start value N on. The O notation talks about the *asymptotic* behavior of the function (when arguments grow towards infinity), therefore at the beginning finitely many exceptions are allowed.

- The bound is independent of concrete *measurement units*, it therefore suffices that it holds *up to a constant* c, which we may choose arbitrarily large. Consequently, if $f(n) = O(g(n))$, then also $c \cdot f(n) = O(g(n))$ for arbitrary c.

This intuitive interpretation of $f(n) = O(g(n))$ is visualized in the first diagram of Figure 5.1.

Analogously, $f(n) = \Omega(g(n))$ expresses the fact that function f "does not grow less" than function g, i.e, that f is "bounded from below" by g (again, towards the infinity and up to a multiplicative constant). Finally $f(n) = \Theta(g(n))$ means that f "grows exactly" like g, i.e, that f is "bounded from above and below" by g. The second and the third diagrams in Figure 5.1 visualize these relationships.

The duality between O and Ω can be also expressed in the following way.

Theorem 39 (Duality of O and Ω) For all $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we have

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Proof We show $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$ by assuming $f(n) = O(g(n))$ and showing $g(n) = \Omega(f(n))$. By the definition of Ω, we have to find constants N_1, c_1 such that

$$\forall n \geq N_1 : g(n) \geq c_1 \cdot f(n)$$

Since $f(n) = O(g(n))$, we have constants N_2, c_2 such that

$$\forall n \geq N_2 : f(n) \leq c_2 \cdot g(n)$$

Take $N_1 := N_2$ and $c_1 := 1/c_2$. Then we have, since $N_1 = N_2$, for all $n \geq N_1$,

$$c_2 \cdot g(n) \geq f(n)$$

and therefore

$$g(n) \geq (1/c_2) \cdot f(n) = c_1 \cdot f(n).$$

The proof of $g(n) = \Omega(f(n)) \Rightarrow f(n) = O(g(n))$ proceeds analogously. □

Example 27 We claim $3n^2 + 5n + 7 = O(n^2)$.

To prove this, we have to find constants c and N such that

$$\forall n \geq N : 3n^2 + 5n + 7 \leq cn^2$$

For $n \geq 1$, we have

$$3n^2 + 5n + 7 \overset{1 \leq n}{\leq} 3n^2 + 5n + 7n = 3n^2 + 12n$$

For $n \geq 12$, we also have

$$3n^2 + 12n \overset{12 \leq n}{\leq} 3n^2 + n \cdot n = 4n^2$$

We thus take $N := 12$ $(= \max\{1, 12\})$ and $c := 4$ and have for all $n \geq N$

$$3n^2 + 5n + 7 \overset{1 \leq n}{\leq} 3n^2 + 5n + 7n = 3n^2 + 12n \overset{12 \leq n}{\leq} 3n^2 + n \cdot n = 4n^2 = cn^2$$ □

This example demonstrated a general technique by which for a function defined by a polynomial expressions all monomials with smaller exponent can be gradually reduced to the monomial with the highest exponent, i.e., $a^m = O(a^{m'})$, for all $m \leq m'$. We thus have a first general result about the asymptotic behavior of a class of complexity functions.

Theorem 40 (Asymptotic Complexity of Polynomial Functions) For all $a_0, \ldots, a_m \in \mathbb{R}$, we have

$$a_m n^m + \ldots + a_2 n^2 + a_1 n + a_0 = \Theta(n^m)$$

i.e., every polynomial function is asymptotically bounded from above and from below by the function denoted by the monomial with the highest exponent.

Proof The proof that the polynomial function is in $O(n^m)$ proceeds along the lines sketched by above example; the proof that it is also in $\Omega(n^m)$ is straight-forward. □

Another result justifies why in asymptotic complexity functions we may simply write $\log n$ rather than $\log_b n$.

Theorem 41 (Logarithms) For arbitrary positive bases $a, b \in \mathbb{R}_{>0}$, we have

$$log_a\ n = O(log_b\ n)$$

Proof We have to find constants c and N such that

$$\forall n \geq N : log_a\ n \leq c \cdot log_b\ n$$

Take $c := \log_a b$ and $N := 0$. Then we have for all $n \geq N$

$$log_a\ n = \log_a(b^{\log_b n}) = (\log_a b) \cdot (\log_b n) = c \cdot (\log_b n)$$ □

The following result exhibits an important relationship between polynomial and exponential functions.

Theorem 42 (Polynomial versus Exponential) For all real constants $a, b \in \mathbb{R}$ with $b > 1$, we have

$$n^a = O(b^n)$$

i.e., every polynomial function with arbitrary high degree is bounded from above by any exponential function with base greater than one.

Proof We have to find constants c and N such that

$$\forall n \geq N : n^a \leq c \cdot b^n$$

We know the Taylor series expansion

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Since $b^n = (e^{\ln b})^n = e^{n \ln b}$, we have for all $n \in \mathbb{N}$

$$b^n = \sum_{i=0}^{\infty} \frac{(n \ln b)^i}{i!} = 1 + (n \ln b) + \frac{(n \ln b)^2}{2!} + \frac{(n \ln b)^3}{3!} + \dots$$

Since $b > 1$, we have $\ln b > 0$; therefore we know

$$b^n > \frac{(n \ln b)^a}{a!} = \frac{(\ln b)^a}{a!} n^a$$

i.e.

$$n^a < \frac{a!}{(\ln b)^a} b^n$$

Thus we define $N := 0$ and $c := a!/(\ln b)^a$ and are done. □

5.3. Working with Asymptotic Complexity

We summarize the asymptotic equations which we have derived so far.

Theorem 43 (Asymptotic Bounds) The following asymptotic bounds hold:

$$\begin{aligned} c \cdot f(n) &= O(f(n)) \\ n^m &= O(n^{m'}), \text{ for all } m \leq m' \\ a_m n^m + \dots + a_2 n^2 + a_1 n + a_0 &= \Theta(n^m) \\ \log_a n &= O(\log_b n), \text{ for all } a, b > 0 \\ n^a &= O(b^n), \text{ for all } a, b \text{ with } b > 1 \end{aligned}$$

Furthermore, the following asymptotic laws can be derived.

Theorem 44 (Asymptotic Laws) Let $f, g, h : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

- **Reflexivity:** $f = O(f)$, $f = \Omega(f)$, $f = \Theta(f)$.
- **Symmetry:**
  - If $f = O(g)$, then $g = \Omega(f)$.
  - If $f = \Omega(g)$, then $g = O(f)$.
  - If $f = \Theta(g)$, then $g = \Theta(f)$.
- **Transitivity:**
  - If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
  - If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
  - If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Proof An easy exercise for the reader. □

As discussed, $f(n) = O(g(n)$ is an abuse of notation, since it actually expresses the property $f \in O(g)$. The following definition sanctions an even more general form of this abuse.

Definition 46 (Asymptotic Notation in Equations) Take an equality of form

$$A[\mathcal{O}_1(f(n))] = B[\mathcal{O}_2(g(n))]$$

where A and B are arbitrary terms with (possibly multiple) occurrences of $\mathcal{O}_1, \mathcal{O}_2 \in \{O, \Omega, \Theta\}$.

Then this equality is to be interpreted as the statement

$$\begin{aligned} &\forall f' \in \mathcal{O}_1(f) : \exists g' \in \mathcal{O}_2(g) : \\ &\forall n \in \mathbb{N} : A[f'(n)] = B[g'(n)] \end{aligned}$$

Every occurrence of an $\mathcal{O}$ term is thus replaced by a function in the corresponding asymptotic complexity class; functions on the left side of the equation are universally quantified, functions on the right side are existentially quantified.

In such an equality, no matter how functions on the left hand side are chosen, there is always a way to chose functions on the right side to make the equation valid.

Example 28 We already encountered in Example 25 the harmonic number H_n. We have the relationship

$$H_n = \ln n + \gamma + O(\frac{1}{n})$$

i.e., there is a function $f \in O(1/n)$ such that, for all $n \in \mathbb{N}$, $H_n = \ln n + \gamma + f(n)$. □

Example 29 The statement

$$2n^2 + 3n + 1 = O(2n^2) + O(n) = O(n^2)$$

expresses a conjunction of the two statements

$$2n^2 + 3n + 1 = O(2n^2) + O(n)$$
$$O(2n^2) + O(n) = O(n^2)$$

which are to be interpreted as

$$\begin{array}{lll} \exists f \in O(2n^2), g \in O(n) : & \forall n \in \mathbb{N} : & 2n^2 + 3n + 1 = f(n) + g(n) \\ \forall f \in O(2n^2), g \in O(n) : \exists h \in O(n^2) : & \forall n \in \mathbb{N} : & f(n) + g(n) = h(n) \end{array}$$

□

With the help of this notation, we can express a number of further relationships.

Theorem 45 (Further Asymptotic Equations) The following asymptotic equations hold:

$$\begin{aligned} O(O(f(n))) &= O(f(n)) \\ O(f(n)) + O(g(n)) &= O(f(n) + g(n)) \\ O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)) \\ O(f(n) \cdot g(n)) &= f(n) \cdot O(g(n)) \\ O(f(n)^m) &= O(f(n))^m, \text{ for all } m \geq 0 \end{aligned}$$

Proof A simple exercise for the reader. □

5.4. Complexity Classes

In this section, we are going to investigate how functions can be categorized according to their asymptotic growth. For this purpose, we will derive some more complexity notions.

Klein-O-Notation
Klein-Omega-Notation

Definition 47 (*Little-o notation*, *Little-ω notation*) Let $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function from $\mathbb{N}$ into the set $\mathbb{R}_{\geq 0}$ of the non-negative real numbers.

- Then $o(g)$ denotes the set of all functions $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

  $$\forall c \in \mathbb{R}_{>0} : \exists N \in \mathbb{N} : \forall n \geq N : f(n) \leq c \cdot g(n)$$

  We write $f(n) = o(g(n))$ (read "$f(n)$ is little oh of $g(n)$") to indicate $f \in o(g)$; we also say that "f is asymptotically smaller than g".

- Then $\omega(g)$ denotes the set of all functions $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that

  $$\forall c \in \mathbb{R}_{>0} : \exists N \in \mathbb{N} : \forall n \geq N : g(n) \leq c \cdot f(n)$$

  We write $f(n) = \omega(g(n))$ (read "$f(n)$ is little omega of $g(n)$") to indicate $f \in \omega(g)$; we also say that "f is asymptotically larger than g".

The definitions are similar to that of O and Ω except that the defining property must hold for *all* values $c \in \mathbb{R}_{\geq 0}$ (rather than for just *some* c). In analogy to O and Ω, also o and ω are dual.

Theorem 46 (Duality of *o* and *ω*) For all $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we have

$$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$$

Proof A simple exercise for the reader. □

The relevance of above definitions becomes clearer by the following theorem.

Theorem 47 (Properties of o and ω) For all $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we have

$$f \in o(g) \Rightarrow f \in O(g) \wedge f \notin \Theta(g)$$
$$f \in \omega(g) \Rightarrow f \in \Omega(g) \wedge f \notin \Theta(g)$$

Proof An exercise for the reader. □

In other words, the notions o and ω are stronger variants of O and Ω: we may think of $f = O(g)$, $f = \Omega(g)$, and $f = \Theta(g)$ as function "comparisons" $f \leq g$, $f \geq g$, and $f = g$. Then $f = o(g)$ and $f = \omega(g)$ may be thought as the strict comparisons $f < g$ and $g < f$, i.e., they exclude the case that f and g have the same asymptotic growth (however, the analogy is incomplete, because the implications in above theorem do not hold in the other direction).

With the help of o, the following theorem (which we state without proof) describes disjoint classes of functions that differ with respect to their asymptotic complexity.

Theorem 48 (Hierarchy of Complexity Classes) Let $f \prec g$ denote $f = o(g)$, i.e., f is asymptotically smaller than g. Then we have the following relationships:

$$\begin{aligned} 1 &\prec \log\log\log n \prec \log\log n \prec \sqrt{\log n} \prec \log n \prec (\log n)^2 \prec (\log n)^3 \\ &\prec \sqrt[3]{n} \prec \sqrt{n} \prec n \prec n \log n \prec n\sqrt{n} \prec n^2 \prec n^3 \\ &\prec n^{\log n} \prec 2^{\sqrt{n}} \prec 2^n \prec 3^n \prec n! \prec n^n \prec 2^{n^2} \prec 2^{2^n} \prec 2^{2^{\cdot^{\cdot^{\cdot^2}}}} \ (n \text{ times}) \end{aligned}$$

Among these classes, the following play a prominent role in computer science.

***O*(1) (Constant)** There is an upper limit on the function values. Such a function may, e.g., denote the space complexity of an algorithm that works with a fixed amount of memory.

***O*(log *n*) (Logarithmic)** The function values grow with the argument, but only very slowly. This describes the runtime of many very efficient algorithms that operate according to the *divide and conquer* principle, e.g., the *binary search* algorithm. Also the class $O((\log n)^k)$ of *polylogarithmic functions* is similarly well-behaved.

***O*(*n*) (Linear)** The function values grow proportionally with the argument. This describes, e.g., the runtime of algorithm that traverse a data structure once, e.g., *linear search*.

***O*(*n* log *n*) (Linear-Logarithmic or *Linearithmic*)** The function values grow more than proportionally with the argument, but the extra growth factor is reasonably well behaved. This describes the runtime of the fast sorting algorithms (average time of Quicksort, worst time of Heapsort and Mergesort). Also the class $O(n(\log n)^c)$ of *quasi-linear* algorithms has similar behavior.

***O*(n^c) (Polynomial)** The function values may grow rapidly but with a polynomial bound. Only algorithms in this class are generally considered as still "feasible" or "tractable" for large inputs, e.g., the multiplication of two matrices with complexity $O(n^3)$.

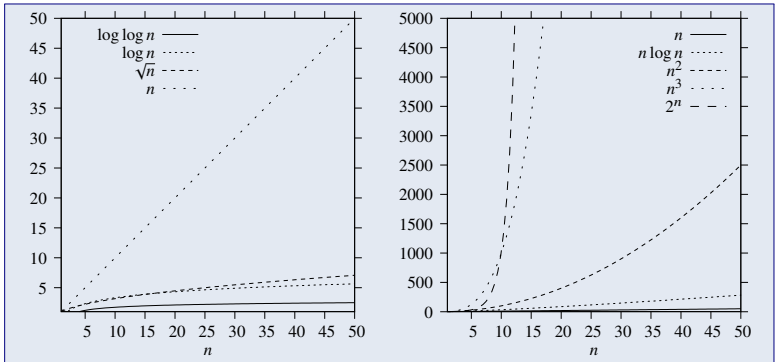
***O*(c^n) (Exponential)** The function values grow extremely rapidly. Problems with such algorithms are in general only solvable for small inputs; this includes the finding of exact solutions to many optimization problems such as the *traveling salesman* problem.

***O*(c^{d^n}) (Double Exponential)** The function values grow overwhelmingly rapidly. While their practical application is limited, various interesting algorithms have this complexity, e.g., decision procedures for statements about real numbers (e.g., *quantifier elimination*), or *Buchberger's algorithm* which can be used to solve multivariate polynomial equations.

Various complexity classes are illustrated in Figure 5.2. The left diagram visualizes complexity functions up to n, the right diagram depicts complexity functions from n up to 2^n (note the different vertical scales). We can see that 2^n is substantially different from polynomial functions in that its slope rapidly grows so much that it seems as if the function would not even be defined for $n > 12$; this justifies to consider computations with worst-case time complexity beyond polynomial as "infeasible" in general (nevertheless, there may be many interesting special cases for which such computations are feasible).

Correspondingly, the table in Figure 5.2 depicts the largest input size n for which an algorithm with the given time complexity can produce a result in one second, one minute, and one hour (assuming that each execution step takes 1 millisecond). We see that the bound grows with a substantial multiplicative factor for all functions up to polynomial time complexity, but only grows by an additive term in the case of an exponential time complexity.

We should note that we can achieve the improvement of giving the algorithm 60 times more time also by investing 60 times more processing power, e.g., by applying a faster processor or (if the algorithm can be parallelized) multiple processors. However, comparing the gains achieved by adding more processing power (going to the right within each line of the table) with the



Time complexity	Maximum n for time bound 1s	1m	1h	Improvement for input size n
n	1000	60000	3600000	$*\ 60$
$n \log n$	140	4895	204094	$*\ 40$ ($\approx$)
n^2	31	244	1897	$*\ 7.75$ ($\sqrt{60}$)
n^3	10	39	153	$*\ 4.91$ ($\sqrt[3]{60}$)
2^n	9	15	21	$+\ 5.9$ ($\log_2 60$)

Figure 5.2.: Complexity Classes

gains achieved by using an algorithm with lower time complexity (going upwards within each column of the table), we see that improvements in algorithmic complexity vastly outperform improvements in processing power; computer science's permanent strive to develop algorithms with even slightly better asymptotic complexity is thus fully justified.

In Chapter 7, we will investigate in more detail the difference between polynomial complexity ("feasible" computations) and higher complexity classes ("infeasible" computations).

Chapter 6.

Analysis of Complexity

In this chapter, we discuss the complexity analysis of several algorithms on the basis of a simple high-level cost model. We start with the discussion of sums (arising in the analysis of iterative algorithms) and recurrences (arising in the analysis of recursive algorithms). Later we investigate the analysis of the important class of *divide and conquer* algorithms and illustrate the analysis of algorithms that use the technique of *randomization*. Finally, we explore the *amortized analysis* of sequences of operations that manipulate some data structure. However, to motivate our further discussions, we first show a small example analysis.

Example 30 Take the program function

```
static int f(int m) {
  if (m == 1) return 1;
  int s = 1;
  for (int i=0; i<log2(m); i++)
     s = s+f(m/2);
  return s;
}
```

which for given argument m calls itself recursively $\lfloor \log_2 m \rfloor$ times with argument $m/2$, i.e, for argument 2^n it calls itself recursively n times with argument 2^{n-1}. We are interested to find out how many times f is called in total (including the calls arising from the recursive invocations); actually this number of invocations is also the result of the function. The function might seem useless, but it can be extended by additional argumens and computations; our analysis will also be valid for such an extended function (provided that the additional computations do not change the number of recursive calls).

The tree of recursive function evaluations for the execution of $f(16) = f(2^4)$ is depicted in Figure 6.1. We see that the root $f(2^4)$ has four children $f(2^3)$ each of which represents the root

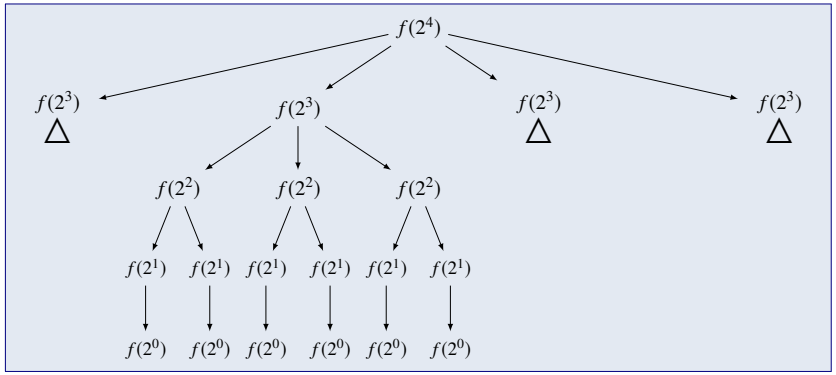


Figure 6.1.: A Recursion Tree

of another subtree (of which only one is shown) with 3 children; each of these 3 children has 2 children each of which has 1 child each of which has 1 child that has no more child.

This tree has height 4, i.e., 5 levels of nodes: at level 0, we have only one node (the root), at level 1 we have 4 nodes, at level 2 with have $4 \cdot 3$ nodes, at level 3 we have $4 \cdot 3 \cdot 2$ nodes, at level 4 we have $4 \cdot 3 \cdot 2 \cdot 1$ nodes. So the total number of nodes in the tree, i.e., the total number of function calls in the execution of $f(16)$, is

$$1 + 4 + 4 \cdot 3 + 4 \cdot 3 \cdot 2 + 4 \cdot 3 \cdot 2 \cdot 1 = 1 + 4 + 12 + 24 + 24 = 65$$

While this example yields some insight, we nevertheless want to determine the number of function calls for arbitrary input m.

To simplify our elaboration, we will assume that m is some power of 2, i.e., $m = 2^n$ for some n. We are now going to determine the number $T(n)$ of function calls in the evaluation of $f(m) = f(2^n)$. From the code, we can easily derive the following recurrence defining $T(n)$:

$$T(n) := \begin{cases} 1 & \text{if } n = 0 \\ 1 + n \cdot T(n-1) & \text{else} \end{cases}$$

For argument $m = 2^0 = 1$, we have only one function call; for argument $m = 2^n > 1$, we have one function call plus $n = \log_2 m$ recursive calls with argument $m/2 = 2^{n-1}$, each of which therefore leads to $T(n-1)$ calls. Our goal is to find an explicit solution to this recurrence.

From the example above, we may get the idea to add the number of nodes in each level of the tree of depth n, i.e.

$$T(n) = 1 + n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \ldots + n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

This summation pattern can be described by the closed formula

$$T(n) = \sum_{i=0}^{n} \frac{n!}{i!}$$

as can be seen from the explicit evaluation of the summands

$$\begin{aligned} n!/n! &= 1 \\ n!/(n-1)! &= n \\ n!/(n-2)! &= n \cdot (n-1) \\ n!/(n-3)! &= n \cdot (n-1) \cdot (n-2) \\ &\cdots \\ n!/0! &= n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \end{aligned}$$

We now want to verify our guess that this sum represents a solution to the recurrence defining $T(n)$. We are therefore going to prove

$$\forall n \in \mathbb{N} : T(n) = \sum_{i=0}^{n} \frac{n!}{i!}$$

by induction on n. For the induction base $n = 0$, the statement clearly holds:

$$T(0) = 1 = 0!/0! = \sum_{i=0}^{0} \frac{0!}{i!}$$

So we assume the induction hypothesis

$$T(n) = \sum_{i=0}^{n} \frac{n!}{i!}$$

and show in the induction step

$$T(n+1) = \sum_{i=0}^{n+1} \frac{(n+1)!}{i!}$$

We then know

$$\begin{aligned} T(n+1) &= 1 + (n+1) \cdot T(n) \\ &= 1 + (n+1) \cdot \sum_{i=0}^{n} \frac{n!}{i!} \\ &= 1 + \sum_{i=0}^{n} \frac{(n+1) \cdot n!}{i!} \\ &= 1 + \sum_{i=0}^{n} \frac{(n+1)!}{i!} \\ &= \frac{(n+1)!}{(n+1)!} + \sum_{i=0}^{n} \frac{(n+1)!}{i!} \\ &= \sum_{i=0}^{n+1} \frac{(n+1)!}{i!} \end{aligned}$$

where the first equation holds because of the defining recurrence and the second equation holds because of the induction hypothesis; thus we have proved our claim.

However, the explicit solution for $T(n)$ does not yield as much insight as we might have hoped for; we will therefore strive for an aymptotic characterization of $T(n)$. We get such a characterization by the derivation

$$T(n) = \sum_{i=0}^{n} \frac{n!}{i!} = n! \cdot \sum_{i=0}^{n} \frac{1}{i!} < n! \cdot \sum_{i=0}^{\infty} \frac{1}{i!} = n! \cdot e = O(n!)$$

where we use the knowledge $\sum_{i=0}^{\infty} 1/i! = e$ (Euler's number). We thus see that the number of invocations of function f grows with the factorial of $n = \log_2 m$. □

6.1. Sums

We start our further considerations with the analysis of Algorithm InsertionSort depicted in Figure 6.2. This algorithm sorts an integer array $a[0 \ldots n-1]$ of length $n \geq 1$ in ascending order by repeatedly inserting element $x = a[i]$ into the already sorted sequence $a[0 \ldots i-1]$ (for every $1 \leq i \leq n-1$). We define as the input size the length n of a, as the time consumption the number of times each statement is executed, and as the space consumption the number of integer values used. It is easy to see that the space complexity of the algorithm is then $S(n) = \overline{S}(n) = n + 4$, because in addition to array a four variables n, i, x, j are used. Our


```
procedure InsertionSort(a)                    Cost
    n ← length(a)                             1
    for i from 1 to n − 1 do                  n
        x ← a[i]                              n − 1
        j ← i − 1                             n − 1
        while j ≥ 0 ∧ a[j] > x do             Σ_{i=1}^{n−1} n_i
            a[j + 1] ← a[j]                   Σ_{i=1}^{n−1} (n_i − 1)
            j ← j − 1                         Σ_{i=1}^{n−1} (n_i − 1)
        end while
        a[j + 1] ← x                          n − 1
    end for
end procedure
```

Figure 6.2.: Insertion Sort

remaining goal is to determine the worst-case time complexity $T(n)$ and the average-case time complexity $\overline{T}(n)$ of InsertionSort.

Worst-case time complexity To the right of Figure 6.2 we depict the number of times each statement is executed where by n_i we understand the maximum number of times that the test of the **while** loop is executed for value i with $1 \leq i \leq n-1$. The total number $T(n)$ of statement executions is thus

$$\begin{aligned} T(n) &= 1 + n + (n-1) + (n-1) + \Big(\sum_{i=1}^{n-1} n_i\Big) + \Big(\sum_{i=1}^{n-1} (n_i - 1)\Big) + \Big(\sum_{i=1}^{n-1} (n_i - 1)\Big) + (n-1) \\ &= 4n - 2 + \sum_{i=1}^{n-1} (3n_i - 2) \end{aligned}$$

In the worst-case scenario, every element $x = a[i]$ has to be inserted at position 0 (this happens when the input a is sorted in *descending* order), thus $n_i = i + 1$ (because $i + 1$ tests $i - 1 \geq 0, \ldots 0 \geq 0, -1 \geq 0$ have to be performed). We thus have

$$\begin{aligned} T(n) &= 4n - 2 + \sum_{i=1}^{n-1} (3 \cdot (i+1) - 2) \\ &= 4n - 2 + \sum_{i=1}^{n-1} (3i + 1) \end{aligned}$$

Our final problem is thus to determine the value of the sum

$$\sum_{i=1}^{n-1}(3i+1)$$

Indeed the core problem of the complexity analysis of algorithms defined in terms of loop iteration is to solve such sums, i.e., to find closed forms for their values (a closed form is a form that does not involve the summation symbol). There exist elaborate methods to solve this problem, but sometimes also high school knowledge suffices. For instance, we should remember at least the closed form solution for an arithmetic series

$$\sum_{i=0}^{n}(a+i\cdot d) = (n+1)\cdot a + d\cdot\frac{n\cdot(n+1)}{2}$$

respectively for a geometric series

$$\sum_{i=0}^{n}(a\cdot q^i) = a\cdot\frac{q^{n+1}-1}{q-1}$$

Since our example above involves just an arithmetic series, we can derive

$$\sum_{i=1}^{n-1}(3i+1) = \sum_{i=0}^{n-1}(3i+1) - 1 = n\cdot 1 + 3\cdot\frac{(n-1)\cdot n}{2} - 1 = \frac{3n^2-n-2}{2}$$

We thus get

$$T(n) = 4n - 2 + \frac{3n^2-n-2}{2} = \frac{3n^2+7n-6}{2}$$

Average-case time complexity To determine the average-case time complexity $\overline{T}(n)$, we need to determine the expected value $E[N_i]$ of the random variable N_i that represents the number of tests of the inner loop for value i. If we assume that all values of a have equal probability, it is not hard to see that we have equal probability to insert $a[i]$ in any of the positions $0,\ldots,i$ and thus equal probability for N_i to have values $1,\ldots,i+1$. We thus get

$$E[N_i] = \frac{1}{i+1}\cdot\sum_{j=1}^{i+1} j = \frac{(i+2)\cdot(i+1)}{2\cdot(i+1)} = \frac{i+2}{2}$$

and thus

$$\begin{aligned}\overline{T}(n) &= 4n - 2 + \sum_{i=1}^{n-1}\left(3\cdot\frac{i+2}{2} - 2\right)\\ &= 4n - 2 + \frac{1}{2}\cdot\sum_{i=1}^{n-1}(3i+2)\end{aligned}$$

where

$$\begin{aligned}\sum_{i=1}^{n-1}(3i+2) &= \sum_{i=0}^{n-1}(3i+2) - 2\\ &= \left(2n + 3\cdot\frac{(n-1)\cdot n}{2}\right) - 2\\ &= \frac{4n+3n^2-3n-4}{2}\\ &= \frac{3n^2+n-4}{2}\end{aligned}$$

We thus have

$$\overline{T}(n) = 4n - 2 + \frac{3n^2+n-4}{4} = \frac{16n-8+3n^2+n-4}{4} = \frac{3n^2+17n-12}{4}$$

For large n, we have $T(n) \simeq \frac{3n^2}{2}$ and $\overline{T}(n) \simeq \frac{3n^2}{4}$ and thus $\overline{T}(n) \simeq \frac{T(n)}{2}$, i.e., the average time complexity of the algorithm is half of its worst case time complexity. However, we have also $T(n) = \overline{T}(n) = \Theta(n^2)$, i.e., the asymptotic time complexity of InsertionSort is quadratic in the length of the input array, both in the worst case and in the average case.

If one is interested just in the asymptotic complexity of InsertionSort, we can get this result much easier by using the approximation

$$\sum_{i=0}^{\Theta(n)}\Theta(i^k) = \Theta(n^{k+1})$$

In other words, if we sum up a number times that is linear in the value of n a value that is polynomial in the summation index with degree k, the result value is polynomial in n with degree $k+1$. Since InsertionSort iterates a linear number of times a loop body of linear complexity, the total complexity of the algorithm is thus quadratic.

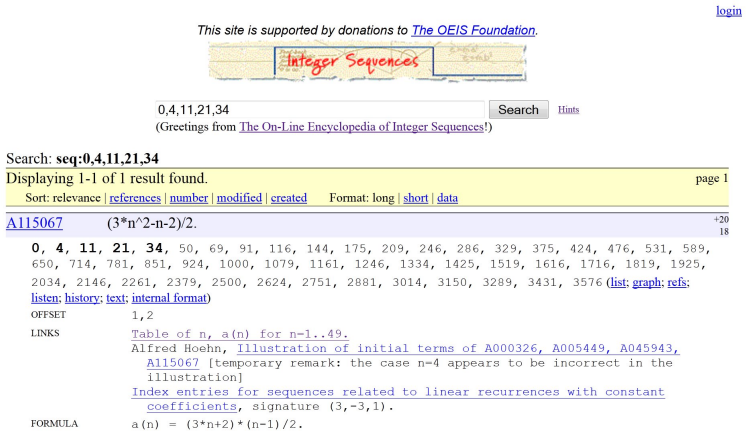


Figure 6.3.: On-Line Encyclopedia of Integer Sequences (OEIS)

Solving Sums by Guessing and Verifying If we are not able to compute a closed form of a sum on our own, there is always the possibility to derive by some means first a *guess* for a solution and then *verify* that this guess is correct.

As for the guessing part, we may go to the library to check some reference book, e.g., the *CRC Standard Mathematical Tables and Formulae* or the *Handbook of Mathematical Functions*, to find a closed solution for a summation problem at hand. Another approach is to determine by hand the first values of the sum, i.e., the sum with 0 summands, with 1 summands, with 2 summands, and so on. In the case of

$$\sum_{i=1}^{n-1}(3i+1)$$

used in the analysis of the worst-case time complexity of InsertionSort, the values of these sums are

$$0, 4, 4+7, 4+7+10, 4+7+10+13, \ldots = 0, 4, 11, 21, 34, \ldots$$

corresponding to $n = 1, 2, 3, 4, 5, \ldots$. We may then consult the *Handbook of Integer Sequences* to determine what is known about this sequence. Even simpler, we may just visit the *On-Line Encyclopedia of Integer Sequences (OEIS)*[1] and type in `0,4,11,21,34` to get, as depicted in

[1] http://oeis.org

Figure 6.3, the answer

$$\frac{3n^2-n-2}{2}$$

which confirms our derivation above.

Rather than consulting tables, we may also harness the power of modern computer algebra systems. We may e.g. enter in the computer algebra system Maple

```
> sum(3*i+1,i=1..n-1);
                                    2
                                3/2 n  - 1/2 n - 1
```

or in the computer algebra system Mathematica

```
In[1]:= Sum[3*i+1,{i,1,n-1}]

                      2
                -2 - n + 3 n
Out[1]= -------------
                2
```

to get the same solution. In Mathematica it is also possible to determine the function directly from the initial values of the integer sequence

```
In[2]:= FindSequenceFunction[{0,4,11,21,34},n]

          (-1 + n) (2 + 3 n)
Out[2]= ------------------
                2
```

similar to the table lookup we have performed above.

No matter how we derived these solutions, by consulting books, tables, or software systems, we should be always wary that the solutions might not be correct (books and tables can and indeed do contain wrong information, software systems can and indeed do have bugs). Thus we need in a second step to verify the solution. This typically leads to a proof by induction.

Example 31 We want to verify the equality

$$\sum_{i=1}^{n-1}(3i+1) = \frac{3n^2-n-2}{2}$$

Actually, a little inspection reveals that this equality does *not* hold for $n = 0$, because

$$\sum_{i=1}^{0-1}(3i+1) = 0 \neq -1 = \frac{3\cdot 0^2-0-2}{2}$$

We skipped over this detail, because in our analysis above we assumed $n \geq 1$ (actually the algorithm also works for $n = 0$; we then have $T(0) = 2$, because only the first two lines of the algorithm are executed). We are therefore going to prove

$$\forall n \in \mathbb{N} : n \geq 1 \Rightarrow \sum_{i=1}^{n-1}(3i+1) = \frac{3n^2 - n - 2}{2}$$

by induction on n.

- Base case $n = 1$:
$$\sum_{i=1}^{1-1}(3i+1) = 0 = \frac{3 \cdot 1^2 - 1 - 2}{2}$$
- We assume for fixed $n \geq 1$
$$\sum_{i=1}^{n-1}(3i+1) = \frac{3n^2 - n - 2}{2}$$
and show
$$\sum_{i=1}^{n}(3i+1) = \frac{3 \cdot (n+1)^2 - (n+1) - 2}{2}$$
We have
$$\begin{aligned}\sum_{i=1}^{n}(3i+1) &= \sum_{i=1}^{n-1}(3i+1) + (3n+1)\\ &= \frac{3n^2 - n - 2}{2} + (3n+1)\\ &= \frac{3n^2 - n - 2 + 6n + 2}{2}\\ &= \frac{3n^2 + 5n}{2}\end{aligned}$$
We also have
$$\frac{3 \cdot (n+1)^2 - (n+1) - 2}{2} = \frac{3n^2 + 6n + 3 - n - 1 - 2}{2} = \frac{3n^2 + 5n}{2}$$
and are therefore done. □


```
function BinarySearch(a, x, l, r)          ▷ n = r − l + 1
    if l > r then                                          1
        return −1                                          1
    end if
    m ← ⌊(l+r)/2⌋                                          1
    if a[m] = x then                                       1
        return m                                           1
    else if a[m] < x then                                  1
        return BinarySearch(a, x, m + 1, r)                ≤ 1 + T(⌊n/2⌋)
    else
        return BinarySearch(a, x, l, m − 1)                ≤ 1 + T(⌊n/2⌋)
    end if
end function
```

Figure 6.4.: Binary Search

6.2. Recurrences

We continue with the analysis of the algorithm BinarySearch depicted in Figure 6.4. This algorithm searches in a sorted array a within index interval $[l, r]$ for the position of value x (if x does not occur at any position of this interval, then the result shall be -1). The algorithm starts with investigating the middle position $m = \lfloor \frac{l+r}{2} \rfloor$. If a holds at this position x, the result is m; if the element at position m is smaller than x, the algorithm continues its search to the right of m by a recursive call with $l = m + 1$; if the element is greater than x, it continues its search to the left of m by a recursive call with $r = m - 1$. If $l > r$, then the search interval is empty, and the algorithm returns -1. Our goal is to determine the worst-case time complexity $T(n)$ and the average-case time complexity $\overline{T}(n)$ of BinarySearch in terms of the size $n = r - l + 1$ of the search interval.

Worst-case time complexity If the search interval is empty, then only the test $l > r$ and the **return** statement are executed, i.e.

$$T(0) = 2$$

However, if the search interval is not empty, then in the worst case we have to sum up the costs of the test $l > r$, of the assignment $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$, of the tests $a[m] = x$ and $a[m] < x$, and of one recursive call of the algorithm on a search interval of size $\lfloor \frac{n}{2} \rfloor$:

$$T(n) = 5 + T\left(\lfloor \frac{n}{2} \rfloor\right)$$

Rekursionsgleichung

All in all, we thus have derived the *recurrence (relation)*

$$T(0) = 2$$
$$T(n) = 5 + T\left(\lfloor \frac{n}{2} \rfloor\right), \text{ if } n \geq 1$$

The core problem of the complexity analysis of algorithms defined in terms of recursion is thus to find closed form solutions for such recurrences. If we assume that $n = 2^m$ for some $m \in \mathbb{N}$, we have $\lfloor \frac{n}{2} \rfloor = 2^{m-1}$ and thus

$$T(2^m) = 5 + T(2^{m-1}) = \underbrace{5 + \ldots + 5}_{m \text{ times}} + T(1) = \underbrace{5 + \ldots + 5}_{m+1 \text{ times}} + T(0) = 5 \cdot (m+1) + 2$$

In other words, a call of the algorithm results in m additional recursive calls before the base case is reached. Since $5 \cdot (m+1) + 2 = 5m + 7$ and $m = \log_2 n$, we have

$$T(n) = 5 \cdot (\log_2 n) + 7$$

It is then not difficult to guess that for arbitrary $n \geq 1$ we have

$$T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$$

(we will verify this guess at the end of this section).

This complexity also considers the case that the element x does not occur in a; if we assume that x occurs in a, the algorithm will never call itself recursively with $n = 0$, i.e., we have only $m - 1$ recursive calls. The modified complexity is then

$$T_{\text{found}}(n) = 5 \cdot \lfloor \log_2 n \rfloor + 2$$

or, in other words, $T_{\text{found}}(n) = T(n) - 5$ (for $n \geq 1$).

Average-case time complexity Let us assume $n = 2^m - 1$, e.g. $n = 15$ and $m = 4$. In this case, the algorithm gives rise to the recursion tree of height $m - 1 = 3$ depicted in Figure 6.5: the label of each node in the tree indicates the position that is investigated by the corresponding function call (initially the position $\frac{0+14}{2} = 7$), the left child of every node denotes the first recursive call in the code of BinarySearch and the right child of a node denotes the second recursive call. Any execution of BinarySearch that finds x is thus depicted by a path in the tree starting at the root; each node of the path represents one recursive invocation; the path ends

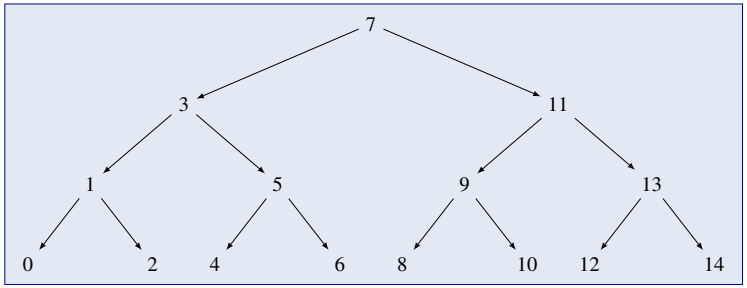


Figure 6.5.: Recursion Tree for BinarySearch

with the node whose index represents the position of x.

Let us assume that x appears at every position with the same probability. If x occurs at position 7, no recursive function call is required to find the element. If x occurs at position 3 or position 11, one recursive call is required; if x occurs at one of the positions $1, 5, 9, 13$, two recursive calls are required; if x occurs at any of the remaining eight positions, three recursive calls are required. Consequently, for each number i from 0 to $m - 1$, there are 2^i array positions that lead to i recursive calls (respectively 2^i paths in the tree that lead to a node at depth i).

Since there are $2^m - 1$ positions (tree nodes) in total, the average number of recursive function calls (lengths of paths in the tree) is thus

$$\frac{1}{2^m - 1} \cdot \sum_{i=0}^{m-1} i \cdot 2^i$$

The value of $\sum_{i=0}^{m-1} i \cdot 2^i$ for $m = 1, 2, 3, 4, 5, \ldots$ is

$$0, 2, 10, 34, 98, \ldots$$

We may guess the value of this sum from the *On-Line Encyclopedia of Integer Sequences*

```
A036799          2+2^(n+1)*(n-1).
         0, 2, 10, 34, 98, ...
```

(please note that the sum here runs until $i = n$, thus for the solution of our problem, we have to substitute $m - 1$ for n). We may also apply Maple

```
> sum(i*2^i,i=0..m-1);
                                        m       m
                                    m 2  - 2 2  + 2
```

to derive the closed form solution

$$\sum_{i=0}^{m-1} i \cdot 2^i = 2^m \cdot (m-2) + 2$$

We leave the proof that this solution is indeed correct as an exercise to the reader.

We thus have for the average number of recursive function calls

$$\frac{1}{2^m - 1} \cdot \sum_{i=0}^{m-1} i \cdot 2^i = \frac{1}{2^m - 1} \cdot (2^m \cdot (m-2) + 2) = \frac{2^m \cdot (m-2)}{2^m - 1} + \frac{2}{2^m - 1}$$

which is, for large m, about $m - 2$. Consequently, we have the average-case time complexity

$$\overline{T}_{\text{found}}(2^m - 1) \simeq 5 \cdot (m-2) + 2 = 5m - 8$$

while the worst-case time complexity is

$$T_{\text{found}}(2^m - 1) = 5 \cdot \lfloor \log_2(2^m - 1) \rfloor + 2 = 5 \cdot (m-1) + 2 = 5m - 3$$

i.e., we have $\overline{T}_{\text{found}}(2^m - 1) \simeq T_{\text{found}}(2^m - 1) - 5$. In other words, for large n the algorithm needs in the average case one recursive call less than in the worst case. This is not surprising, since of the $2^m - 1$ elements in a, already $2^{m-1} - 1$ elements (approximately the half) are detected by the inner nodes of the call tree, not by its leaves.

If we drop the assumption that x occurs in a, we have to consider 2^{m-1} additional cases where one more recursive call is required to detect that x does not occur in a. Assuming that each of these cases is as likely as finding x in a, we have the average number of recursive function calls

$$\begin{aligned}
\frac{1}{2^m - 1 + 2^{m-1}} \cdot \Big((\sum_{i=0}^{m-1} i \cdot 2^i) + 2^{m-1} \cdot m \Big) &= \frac{2^m \cdot (m-2) + 2 + 2^{m-1} \cdot m}{2^m - 1 + 2^{m-1}} \\
&= \frac{2^{m-1} \cdot (2m - 4 + m) + 2}{2^{m-1} \cdot (2+1) - 1} \\
&= \frac{2^{m-1} \cdot (3m-4) + 2}{2^{m-1} \cdot 3 - 1} \\
&\leq \frac{2^{m-1} \cdot 3 \cdot (m-1) + 2}{2^{m-1} \cdot 3 - 1}
\end{aligned}$$

which is, for large m, about $m - 1$. Compared to the worst case with m recursive calls, we have

also here one recursive call less and get the average-case time complexity

$$\overline{T}(2^m - 1) \approx 5 \cdot (m-1) + 2 = 5m - 3$$

while the worst-case time complexity is

$$T(2^m - 1) = 5 \cdot \lfloor \log_2(2^m - 1) \rfloor + 7 = 5 \cdot (m-1) + 7 = 5m + 2$$

We thus also have $\overline{T}(2^m - 1) \simeq T(2^m - 1) - 5$, i.e., save one recursive call in the average case.

The analysis of the average case behavior for an input with size $2^m - 1 < n < 2^{m+1} - 1$ is more complex, because some leaves of the corresponding call tree occur at depth $m - 1$ and some at depth m. However, it is not difficult to see that this time complexity is bounded on the lower end by the average case behavior for a call tree with $2^m - 1$ nodes and on the upper end by the average case behavior for a call tree with $2^{m+1} - 1$ nodes. Since the difference is only a single recursive function call with a cost of 5 units, we refrain from a more detailed analysis of the general case.

Solving Recurrences by Guessing and Verifying In our analysis of the worst-case time complexity of Algorithm BinarySearch, we had to solve the recurrence

$$\begin{aligned}
T(0) &= 2 \\
T(n) &= 5 + T(\lfloor \frac{n}{2} \rfloor), \text{ if } n \geq 1
\end{aligned}$$

As shown below, it is for this recurrence technically easier to use the base case $n = 1$, i.e.,

$$\begin{aligned}
T(1) &= 7 \\
T(n) &= 5 + T(\lfloor \frac{n}{2} \rfloor), \text{ if } n > 1
\end{aligned}$$

The solutions of T for $n = 1, 2, 3, 4, 5, 6, 7, 8, \ldots$ are

$$7, 12, 12, 17, 17, 17, 17, 22, \ldots$$

for which the *On-Line Encyclopedia of Integer Sequences* does unfortunately not give a result. However, this is not very surprising, because the recurrence involves the special coefficients 7 and 5 for which there might be arbitrary other choices. Taking the related recurrence with

simpler coefficients

$$U(1) = 0$$
$$U(n) = 1 + U(\lfloor \frac{n}{2} \rfloor), \text{ if } n > 1$$

with solutions for $n = 1, 2, 3, 4, 5, 6, 7, 8, \ldots$

$$0, 1, 1, 2, 2, 2, 2, 3, \ldots$$

we get the result

```
A000523  Log_2(n) rounded down.
        0, 1, 1, 2, 2, 2, 2, 3, ...
        ...
        FORMULA ... a(n)=floor(lb(n)).
```

i.e., $U(n) = \lfloor \log_2 n \rfloor$. Since T and U only differ in the coefficients, this should give a hint to look for a solution of form $T(n) = a \cdot \lfloor \log_2 n \rfloor + b$. From $T(1) = 7$, we get $b = 7$; from $T(2) = 12$, we get $a = 5$.

Differenzengleichung

q-Differenzengleichung

We may also try the power of computer algebra systems, which are usually able to deal with recurrences of form $T(n) = \ldots T(n-a) \ldots$ (a *difference equation* where additionally the boundary value $T(0)$ must be specified) or $T(n) = \ldots T(n/q) \ldots$ (a *q-difference equation* where additionally the boundary value $T(1)$ must be specified). In Maple, we thus get

```
> rsolve({T(1)=7,T(n)=5+T(n/2)},T(n));
                                7 ln(2) + 5 ln(n)
                                -----------------
                                      ln(2)
```

while Mathematica tells us

```
In[3]:= RSolve[{T[1]==7,T[n]==5+T[n/2]},T[n],n]

                          5 Log[n]
Out[3]= {{T[n] -> 7 + --------}}
                           Log[2]
```

Thus both systems give us the result $T(n) = 5 \cdot (\log_2 n) + 7$. While this is a function with real values, we may nevertheless guess the corresponding solution over the domain of natural numbers as $T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$.

No matter how we have derived our guess, it ultimately has to be verified.

Example 32 We have to show that $5 \cdot \lfloor \log_2 n \rfloor + 7$ is a solution to the recurrence

$$T(1) = 7$$
$$T(n) = 5 + T(\lfloor \frac{n}{2} \rfloor), \text{ if } n > 1$$

i.e, we have to show for all $n \in \mathbb{N}$ with $n \geq 1$,

$$T(n) = 5 \cdot \lfloor \log_2 n \rfloor + 7$$

We prove this by induction on n. For the induction base $n = 1$, we have

$$5 \cdot \lfloor \log_2 1 \rfloor + 7 = 5 \cdot 0 + 7 = 7 = 5 + 2 = 5 + T(0) = 5 + T(\lfloor \frac{1}{2} \rfloor) = T(1)$$

Now assume $n > 1$. By the induction hypothesis, we assume for all m with $1 \leq m < n$

$$T(m) = 5 \cdot \lfloor \log_2 m \rfloor + 7$$

First, we assume n is even, i.e., $n = 2m$ for some $m \in \mathbb{N}$. We then have

$$\begin{aligned} T(n) &= 5 + T(\lfloor \frac{n}{2} \rfloor) = 5 + T(m) = 5 + (5 \cdot \lfloor \log_2 m \rfloor + 7) = 5 \cdot (1 + \lfloor \log_2 m \rfloor) + 7 \\ &= 5 \cdot \lfloor 1 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 7 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 2m \rfloor + 7 \\ &= 5 \cdot \lfloor \log_2 n \rfloor + 7 \end{aligned}$$

Second, we assume n is odd, i.e., $n = 2m + 1$ for some $m \in \mathbb{N}$. We then have

$$\begin{aligned} T(n) &= 5 + T(\lfloor \frac{n}{2} \rfloor) = 5 + T(m) = 5 + (5 \cdot \lfloor \log_2 m \rfloor + 7) = 5 \cdot (1 + \lfloor \log_2 m \rfloor) + 7 \\ &= 5 \cdot \lfloor 1 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 2 + \log_2 m \rfloor + 7 = 5 \cdot \lfloor \log_2 2m \rfloor + 7 \\ &= 5 \cdot \lfloor \log_2 (n-1) \rfloor + 7 = 5 \cdot \lfloor \log_2 n \rfloor + 7 \end{aligned}$$

where the last but one equality holds, because $n > 1$ and n is odd. □

From the proof above, it is easy to see that for arbitrary a, b with $a > 0$, for a recurrence

$$T(1) = b$$
$$T(n) = a + T(\lfloor \frac{n}{2} \rfloor), \text{ if } n > 1$$

```
procedure MergeSort(a, l, r)        ▷ n = r − l + 1        Cost
    if l < r then                                          1
        m ← ⌊(l+r)/2⌋                                      1
        MergeSort(a, l, m)                                 1 + T(⌈n/2⌉)
        MergeSort(a, m + 1, r)                             1 + T(⌊n/2⌋)
        Merge(a, l, m, r)                                  O(n)
    end if
end procedure
```

Figure 6.6.: Mergesort

we have the solution

$$T(n) = a \cdot \lfloor \log_2 n \rfloor + b$$

In the following section, we will deal with more general recurrences of this kind.

6.3. Divide and Conquer

Teile und Herrsche

In this section, we investigate the analysis of algorithms that follow the *divide and conquer* principle: they divide their input into a number of smaller parts, call themselves recursively to solve the problem for each part, and then combine the partial results to the overall results. Since the algorithms are recursive, they give rise to recurrences that have to be solved. However, unlike in the previous section, where we took painstaking care of the derivation of their exact complexity functions, we will now be content with the derivation of asymptotic bounds for these functions. Such asymptotic bounds can be much easier derived and are in the practice of algorithm analysis often the only quantities of real interest.

Sorting We start with the analysis of Algorithm MergeSort depicted in Figure 6.6: this algorithm sorts an array a in the index interval $[l, r]$ of size $n = r - l + 1$. The algorithm calls itself twice recursively to sort a in subintervals $[l, m]$ and $[m + 1, r]$, where m is the middle position in the interval, and then calls an Algorithm Merge that merges the sorted parts in linear time to the totally sorted array. From the algorithm, we can derive the recurrence

$$\begin{aligned} T(n) &= 1 + 1 + 1 + T\left(\lceil \tfrac{n}{2} \rceil\right) + 1 + T\left(\lfloor \tfrac{n}{2} \rfloor\right) + O(n) \\ &= T\left(\lceil \tfrac{n}{2} \rceil\right) + T\left(\lfloor \tfrac{n}{2} \rfloor\right) + O(n) + 4 \\ &= T\left(\lceil \tfrac{n}{2} \rceil\right) + T\left(\lfloor \tfrac{n}{2} \rfloor\right) + O(n) \end{aligned}$$

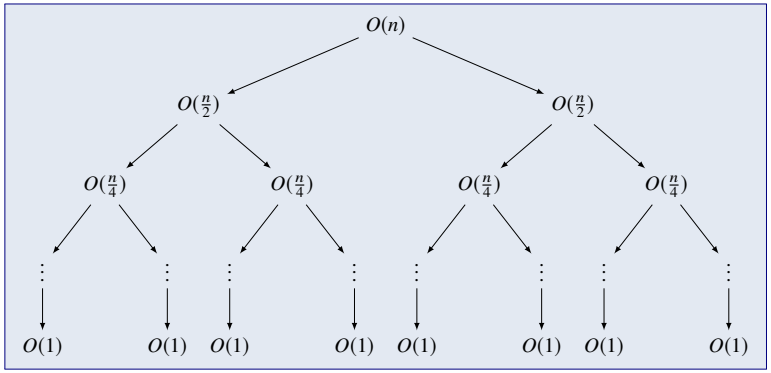


Figure 6.7.: Recursion Tree for MergeSort

Here we only write the definition for $n > 1$ with the implicit understanding that the base case $n = 1$ can be solved in constant time $O(1)$. We can simplify this recurrence further to

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

by writing $\frac{n}{2}$ rather than $\lfloor \frac{n}{2} \rfloor$ respectively $\lceil \frac{n}{2} \rceil$, because the difference does not really play a role in asymptotic analysis (we further on assume that $\frac{n}{2}$ denotes any of these numbers).

To guess a solution of this recurrence, we investigate the recursion tree of height $O(\log n)$ depicted in Figure 6.7 whose nodes are labeled with the asymptotic costs that occur in each call. At depth 0, we have a single cost $O(n)$, at depth 1, we have twice the costs $O(\frac{n}{2})$, at depth 2, we have four times the costs $O(\frac{n}{4})$, and so on. At the leaves of the tree, we have n times the costs $O(1)$. The execution of MergeSort is depicted by a depth-first left-to-right traversal of all nodes in the tree, i.e., the complexity of the algorithm is denoted by the sum of all labels. We estimate the asymptotic worst-case time complexity of MergeSort thus as

$$T(n) = \sum_{i=0}^{O(\log n)} 2^i \cdot O\left(\frac{n}{2^i}\right) = \sum_{i=0}^{O(\log n)} O(n) = O(n \cdot \log n)$$

Actually, for the more special form of the recurrence

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n, & \text{otherwise} \end{cases}$$

we can easily prove the exact solution

$$\forall i \in \mathbb{N} : T(2^i) = 2^i \cdot (i+1)$$

by induction on i:

- Base case: we have $T(2^0) = T(1) = 1 = 2^0 \cdot 1$.
- Induction assumption: we assume $T(2^i) = 2^i \cdot (i+1)$.
- Induction step: we have

$$\begin{aligned} T(2^{i+1}) &= 2 \cdot T(2^i) + 2^{i+1} = 2 \cdot 2^i \cdot (i+1) + 2^{i+1} \\ &= 2^{i+1} \cdot (i+1) + 2^{i+1} = 2^{i+1} \cdot (i+2). \end{aligned}$$

For $n = 2^i$, we thus have $T(n) = n \cdot (\log_2 n + 1) = O(n \cdot \log n)$.

In the following, we will verify the general estimation.

Proof We prove that every solution $T(n)$ of the recurrence

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$$

is asymptotically bound by $T(n) = O(n \cdot \log n)$. From the recurrence and the definition of $O(n)$, we know that there exist some $c > 0$ and $N \geq 1$ such that, for all $n \geq N$

$$T(n) \leq 2 \cdot T(\frac{n}{2}) + c \cdot n$$

From the definition of $O(n \cdot \log n) = O(n \cdot \log_2 n)$, it suffices to show

$$\exists c \in \mathbb{R}_{>0}, N \in \mathbb{N} : \forall n \geq N : T(n) \leq c \cdot n \cdot \log_2 n$$

Thus our goal is to find suitable c' and N' for which we are able to prove

$$\forall n \geq N' : T(n) \leq c' \cdot n \cdot \log_2 n$$

Because of the characterization of T by a recurrence, this naturally leads to an induction proof with base N'. The choice of N' must be carefully considered. Clearly $N' \geq 2$, because for $n = 1$, we have $c' \cdot n \cdot \log_2 n = 0$ which invalidates our goal.

However, a choice of $N' \geq 2$ is problematic for a recurrence of form $T(n) = \ldots T(\frac{n}{2}) \ldots$, because the induction step will prove the goal for n based on the induction assumption that the goal holds for $\frac{n}{2}$. This is not a problem for $N' = 1$, because every sequence of divisions by 2 which starts with a number $n \geq 1$ eventually leads to the base case $n = 1$. However, if $N' \geq 2$ such a sequence may bypass the base case $n = N'$, e.g., we have $(N'+1)/2 < N'$. We therefore have to show the goal not only for the base case $n = N'$ but for all base cases n with $2 \leq n \leq N'$; furthermore, we must ensure that every sequence of divisions starting with a value $n > N$ eventually reaches one of these base cases, i.e., that $\frac{n}{2} \geq 2$. Since this is ensured for $N' \geq 3$, we choose

$$N' := \max\{\sqcup, 3, N\}$$

where $\sqcup$ is a quantity that will be determined in the course of the proof.

As for the choice of c, we have to consider that we have to prove

$$T(n) \leq c' \cdot n \cdot \log_2 n$$

for all $2 \leq n \leq N'$. Fortunately, for all these n, $n \cdot \log_2 n \geq 1$ holds, thus it suffices to prove $T(n) \leq c'$. Furthermore, since $T(2) \leq T(3) \leq \ldots \leq T(N')$, it suffices to prove

$$T(N') \leq c'$$

Still c' might not be large enough. We thus define

$$c' := \max\{\sqcup, T(N')\}$$

where $\sqcup$ is a quantity that will be determined in the course of the proof.

After our initial considerations about the appropriate choice of N' and c', we now proceed to the induction proof itself. By our choice, we have already ensured for all $2 \leq n \leq N'$

$$T(n) \leq c' \cdot n \cdot \log_2 n$$

i.e., the goal holds for all base cases of the induction. To show in the induction step the goal holds also for $n > N'$, we assume for all $2 \leq m < n$

$$T(m) \leq c' \cdot m \cdot \log_2 m$$

and show

$$T(n) \le c' \cdot n \cdot \log_2 n$$

If we interpret $\frac{n}{2}$ as $\lfloor \frac{n}{2} \rfloor$, then we have, since $n > N' \ge N$,

$$\begin{aligned} T(n) &= 2 \cdot T\big(\lfloor \frac{n}{2} \rfloor\big) + c \cdot n \\ &\le 2 \cdot c' \cdot \lfloor \frac{n}{2} \rfloor \cdot \log_2 \lfloor \frac{n}{2} \rfloor + c \cdot n \\ &\le 2 \cdot c' \cdot \frac{n}{2} \cdot \log_2\big(\frac{n}{2}\big) + c \cdot n \\ &= c' \cdot n \cdot \log_2\big(\frac{n}{2}\big) + c \cdot n \\ &= c' \cdot n \cdot ((\log_2 n) - 1) + c \cdot n \\ &= c' \cdot n \cdot (\log_2 n) + (c - c') \cdot n \le c' \cdot n \cdot \log_2 n \end{aligned}$$

where the last inequality holds if $c' \ge c$, because then $c - c' \le 0$.

If we interpret $\frac{n}{2}$ as $\lceil \frac{n}{2} \rceil$, then we have, since $n > N' \ge N$,

$$\begin{aligned} T(n) &= 2 \cdot T\big(\lceil \frac{n}{2} \rceil\big) + c \cdot n \\ &\le 2 \cdot c' \cdot \lceil \frac{n}{2} \rceil \cdot \log_2 \lceil \frac{n}{2} \rceil + c \cdot n \\ &\le c' \cdot (n+1) \cdot \log_2\big(\frac{n+1}{2}\big) + c \cdot n \\ &= c' \cdot (n+1) \cdot (\log_2(n+1) - 1) + c \cdot n \\ &\le c' \cdot (n+1) \cdot ((\log_2 n) + \frac{1}{2} - 1) + c \cdot n \end{aligned}$$

where in the last line we use the inequality $\log_2(n+1) \le (\log_2 n) + \frac{1}{2}$ which holds for all $n > 2$ and thus for all $n \ge N'$ (a fact that has to be established by a separate proof which we omit). We continue the derivation with

$$\begin{aligned} T(n) &\le c' \cdot (n+1) \cdot \big((\log_2 n) + \frac{1}{2} - 1\big) + c \cdot n \\ &= c' \cdot (n+1) \cdot \big((\log_2 n) - \frac{1}{2}\big) + c \cdot n \\ &= c' \cdot n \cdot (\log_2 n) + \frac{c'}{2} \cdot (2 \cdot (\log_2 n) - n - 1) + c \cdot n \\ &\le c' \cdot n \cdot (\log_2 n) + \frac{c'}{2} \cdot (2 \cdot \frac{n}{4} - n - 1) + c \cdot n \end{aligned}$$

where we use in the last line $\log_2 n \le \frac{n}{4}$, which holds for all $n \ge 16$ (which has to be established by a separate proof which we omit); thus we must ensure $N' \ge 16$. We continue with

$$\begin{aligned} T(n) &\le c' \cdot n \cdot (\log_2 n) + \frac{c'}{2} \cdot (2 \cdot \frac{n}{4} - n - 1) + c \cdot n \\ &\le c' \cdot n \cdot (\log_2 n) - \frac{c' \cdot n}{4} + c \cdot n \\ &= c' \cdot n \cdot (\log_2 n) + \frac{4c - c'}{4} \cdot n \le c' \cdot n \cdot \log_2 n \end{aligned}$$

where the last inequality holds if $c' \ge 4c$, because then $4c - c' \le 0$.

From the analysis of both cases above, we know that $N' \ge 16$ and $c' \ge 4c = \max\{c, 4c\}$ suffices to make the proof succeed and thus define $N' := \max\{16, N\}$ and $c' := \max\{4c, T(N')\}$.□

The Master Theorem The result derived above is actually a consequence of the following theorem which allows us to determine asymptotic bounds for the solutions of recurrences that arise from a large class of divide and conquer algorithms, without requiring further proof of the correctness of the bounds.

Hauptsatz der Laufzeitfunktionen

Theorem 49 (*Master theorem*) Let $a \ge 1$, $b > 1$, and $f : \mathbb{N} \to \mathbb{N}$. If $T : \mathbb{N} \to \mathbb{N}$ satisfies the recurrence

$$T(n) = a \cdot T\big(\frac{n}{b}\big) + f(n)$$

where $\frac{n}{b}$ means either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$, then we have the following results:

- If $f(n) = O(n^{(\log_b a) - \varepsilon})$ for some $\varepsilon > 0$, then
$$T(n) = \Theta(n^{\log_b a})$$
- If $f(n) = \Theta(n^{\log_b a})$, then
$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$
- If $f(n) = \Omega(n^{(\log_b a) + \varepsilon})$ for some $\varepsilon > 0$ and there exist some c with $0 < c < 1$ and some $N \in \mathbb{N}$ such that
$$\forall n \ge N : a \cdot f\big(\frac{n}{b}\big) \le c \cdot f(n)$$
then
$$T(n) = \Theta(f(n))$$

In each of the three cases of this theorem, the function f is compared with the function $n^{\log_b a}$; in essence, it is the larger function that determines the solution of the recurrence. If f grows substantially slower, then the first case applies, and the solution of the recurrence is of the order of $n^{\log_b a}$. If both functions have the same asymptotic growth, then the second case applies, and the solution is of the order $n^{\log_b a}$ multiplied by a logarithmic factor. It f grows substantially faster (and f satisfies some side condition), then the third case applies, and the solution is of the order of f. By the term "substantially" in the first respectively third case, we mean that the growth rate of f must be *polynomially* smaller respectively larger than $n^{\log_b a}$, i.e., the function must grow, for some $\varepsilon > 0$, by a factor of n^{ε} asymptotically slower respectively faster than $n^{\log_b a}$. This leaves some "gaps" between cases 1 and 2 respectively between cases 2 and 3 where the theorem cannot be applied.

The proof of the theorem is based on the analysis of the recursion tree determined by the recurrence $T(n) = a \cdot T(\frac{n}{b}) + f(n)$; this tree of arity a has depth $\log_b n$ and each of the a^i nodes at depth i has cost $f(\frac{n}{b^i})$. The total cost of the tree is thus

$$\sum_{i=0}^{\log_b n} a^i \cdot f(\frac{n}{b^i})$$

The three cases of the theorem correspond to those cases where (1) this sum is dominated by the cost of level $i = \log_b n$ (the leaves of the tree), (2) the sum is evenly determined by all levels of the tree, or (3) the sum is dominated by by the cost of level $i = 0$ (the root of the tree); we omit the details of the analysis.

Example 33 For the solution of the recurrence

$$T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$$

we can apply the second case of the master theorem with $a = b = 2$, because then $\log_b a = 1$ and thus $\Theta(n) = \Theta(n^1) = \Theta(n^{\log_b a})$. We thus have

$$T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^1 \cdot \log n) = \Theta(n \cdot \log n)$$

If we assume that the asymptotic time complexity of Merge is $\Theta(n)$, we have thus determined the asymptotic time complexity of MergeSort. □

Arbitrary Precision Multiplication As another example, let us consider the problem of multiplying two natural numbers a and b with n digits each, which gives as a result a number c


```
function Multiply(a, b)                          ▷ digits(a) = digits(b)
    n ← digits(a)                                                  Cost
    c ← 0                                                          1
    for i from n − 1 to 0 do                                       1
        p ← MultiplyDigit(a, b_i)                                  n + 1
        c ← Shift(c, 1)                                            n · Θ(n)
        c ← Add(p, c)                                              Σ_{i=0}^{n−1} Θ(2n − i − 1)
    end for                                                        n · Θ(n)
    return c                                                       1
end function
```

Figure 6.8.: Arbitrary Precision Multiplication (Iterative)

with at most $2n$ digits; the numbers are represented in a positional number system with some base d. The classical multiplication algorithm taught in school proceeds in n iterations (see Figure 6.8). In iteration i, number a is multiplied with digit i of b where digit 0 is the least significant digit and digit $n - 1$ the most significant one; the resulting product p is a number with at most $n + 1$ digits. We then shift the current value of c (a number with at most $2n - i - 1$ digits) by one digit and add p to get the next value of c. The total time complexity thus is

$$T(n) = 3 + (n + 1) + 2n \cdot \Theta(n) + \sum_{i=0}^{n-1} \Theta(2n - i - 1) = \Theta(n^2)$$

i.e., it is quadratic in the number of digits.

Now let us attempt a *recursive* solution to the problem following the divide and conquer principle. If we assume that n is even, we can split the sequence of the n digits of a into two subsequences of $\frac{n}{2}$ digits which represent two numbers a' (consisting of the more significant digits of a) and a'' (consisting of the less significant digits); analogously we split b into two subsequences which represent numbers b' and b''. We then have

$$a = a' \cdot d^{\frac{n}{2}} + a''$$
$$b = b' \cdot d^{\frac{n}{2}} + b''$$

because shifting the first subsequence by $\frac{n}{2}$ digits to the left and filling up the gap with the second subsequence gives the original sequence. We then have

$$\begin{aligned} a \cdot b &= (a' \cdot d^{\frac{n}{2}} + a'') \cdot (b' \cdot d^{\frac{n}{2}} + b'') \\ &= a' \cdot b' \cdot d^n + (a' \cdot b'' + a'' \cdot b') \cdot d^{\frac{n}{2}} + a'' \cdot b'' \end{aligned}$$

```
function Multiply(a, b)                                          Cost
    n ← digits(a)      ▷ digits(a) = digits(b) = 2^m             1
    if n = 1 then                                                1
        c ← MultiplyDigit(a_0, b_0)                              Θ(1)
    else
        a' ← a_{n/2...n-1}; a'' ← a_{0...n/2-1}                  Θ(n)
        b' ← b_{n/2...n-1}; b'' ← b_{0...n/2-1}                  Θ(n)
        u ← Multiply(a', b')                                     1 + T(n/2)
        v ← Multiply(a', b'')                                    1 + T(n/2)
        w ← Multiply(a'', b')                                    1 + T(n/2)
        x ← Multiply(a'', b'')                                   1 + T(n/2)
        y ← Add(v, w)                                            Θ(n)
        y ← Shift(y, n/2)                                        Θ(n)
        c ← Shift(u, n)                                          Θ(n)
        c ← Add(c, y)                                            Θ(n)
        c ← Add(c, x)                                            Θ(n)
    end if
    return c                                                     1
end function
```

Figure 6.9.: Arbitrary Precision Multiplication (Recursive)

Consequently, to compute the product $a \cdot b$ of two numbers of length n, we need to perform 4 multiplications $a' \cdot b'$, $a' \cdot b''$, $a'' \cdot b'$ and $a'' \cdot b''$ of numbers of length $\frac{n}{2}$ and a couple of additions and shift operations. The corresponding algorithm is depicted in Figure 6.9 (we assume for simplicity that n is a power of 2 such that repeated division by 2 always gives an exact result).

The time complexity of this algorithm is determined by the recurrence

$$\begin{aligned} T(n) &= 3 + 4 \cdot \left(1 + T(\frac{n}{2})\right) + 7 \cdot \Theta(n) \\ &= 4 \cdot T(\frac{n}{2}) + \Theta(n) \end{aligned}$$

To solve this recurrence, we can apply the master theorem for $a = 4$ and $b = 2$. We then have $(\log_b a) = (\log_2 4) = 2$. Since $f(n) = O(n) = O(n^1) = O(n^{2-1}) = O(n^{(\log_b a)-1})$, the first case of the theorem applies, and we have

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

Consequently also the time complexity of the recursive algorithm is quadratic in the number of digits. Since this algorithm is also more complicated than the iterative algorithm, we therefore

do not really have any incentive to apply the recursive version. For a long time, it was even conjectured that there is no algorithm for arbitrary precision multiplication that is asymptotically faster than the school algorithm.

However, in 1962 Anatolii Karatsuba and Yuri Ofman realized that the equation derived above can be transformed one step further

$$\begin{aligned} a \cdot b &= (a' \cdot d^{\frac{n}{2}} + a'') \cdot (b' \cdot d^{\frac{n}{2}} + b'') \\ &= a' \cdot b' \cdot d^n + (a' \cdot b'' + a'' \cdot b') \cdot d^{\frac{n}{2}} + a'' \cdot b'' \\ &= a' \cdot b' \cdot d^n + \big((a' + a'') \cdot (b' + b'') - a' \cdot b' - a'' \cdot b''\big) \cdot d^{\frac{n}{2}} + a'' \cdot b'' \end{aligned}$$

Consequently, we can compute $a \cdot b$ by only 3 multiplications $a' \cdot b'$, $(a' + a'') \cdot (b' + b'')$, and $a'' \cdot b''$. However, the numbers $s = a' + a''$ and $t = b' + b''$ may be $\frac{n}{2} + 1$ digit numbers, i.e.,

$$\begin{aligned} s &= s_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + s' \\ t &= t_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + t' \end{aligned}$$

where s' and t' represent the lower $\frac{n}{2}$ digits of s and t. But since we have

$$\begin{aligned} s \cdot t &= (s_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + s') \cdot (t_{\frac{n}{2}} \cdot d^{\frac{n}{2}} + t') \\ &= s_{\frac{n}{2}} \cdot t_{\frac{n}{2}} \cdot d^n + (s_{\frac{n}{2}} \cdot t' + t_{\frac{n}{2}} \cdot s') \cdot d^{\frac{n}{2}} + s' \cdot t' \end{aligned}$$

we can even in this case compute $s \cdot t$ by a single product $s' \cdot t'$ of numbers of length $\frac{n}{2}$ and a couple of additional operations of time complexity $\Theta(n)$.

The corresponding algorithm is depicted in Figure 6.10 (again we assume for simplicity that n is a power of 2) which makes use of only three recursive calls (as explained above, the second call has to deal with the extra complexity of multiplying $\frac{n}{2} + 1$ digit numbers). The time complexity of this algorithm is determined by the recurrence

$$\begin{aligned} T(n) &= 3 + 2 \cdot \left(1 + \cdot T(\frac{n}{2})\right) + \left(\Theta(n) + T(\frac{n}{2})\right) + 10 \cdot \Theta(n) \\ &= 3 \cdot T(\frac{n}{2}) + \Theta(n) \end{aligned}$$

To solve this recurrence, we can apply the master theorem for $a = 3$ and $b = 2$. We then have $\log_b a = \log_2 3$ where $1.58 < \log_2 3 < 1.59$. Since $f(n) = O(n) = O(n^1) = O(n^{(\log_b a)-\varepsilon})$ for $\varepsilon = (\log_2 3) - 1 > 0.58$, the first case of the theorem applies, and we have

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

	Cost
function MULTIPLY(a, b)	
$n \leftarrow digits(a)$ ▷ $digits(a) = digits(b) = 2^m$	1
if $n = 1$ **then**	1
$c \leftarrow$ MULTIPLYDIGIT(a_0, b_0)	$\Theta(1)$
else	
$a' \leftarrow a_{\frac{n}{2}\ldots n-1}; a'' \leftarrow a_{0\ldots\frac{n}{2}-1}$	$\Theta(n)$
$b' \leftarrow b_{\frac{n}{2}\ldots n-1}; b'' \leftarrow b_{0\ldots\frac{n}{2}-1}$	$\Theta(n)$
$s \leftarrow$ ADD(a', a'')	$\Theta(n)$
$t \leftarrow$ ADD(b', b'')	$\Theta(n)$
$u \leftarrow$ MULTIPLY(a', b')	$1 + T(\frac{n}{2})$
$v \leftarrow$ MULTIPLY'(s, t)	$\Theta(n) + T(\frac{n}{2})$
$x \leftarrow$ MULTIPLY(a'', b'')	$1 + T(\frac{n}{2})$
$y \leftarrow$ SUBTRACT(v, u)	$\Theta(n)$
$y \leftarrow$ SUBTRACT(v, x)	$\Theta(n)$
$y \leftarrow$ SHIFT$(y, \frac{n}{2})$	$\Theta(n)$
$c \leftarrow$ SHIFT(u, n)	$\Theta(n)$
$c \leftarrow$ ADD(c, y)	$\Theta(n)$
$c \leftarrow$ ADD(c, x)	$\Theta(n)$
end if	
return c	1
end function	

Figure 6.10.: Arbitrary Precision Multiplication (Karatsuba Algorithm)

Since $(\log_2 3) < 2$, we have $\Theta(n^{\log_2 3}) = o(n^2)$, i.e., the Karatsuba algorithm performs asymptotically better than the classical multiplication algorithm. Software systems that have to perform arithmetic with arbitrarily long integers (such as computer algebra systems) indeed implement this algorithm.

6.4. Randomization

We are going to analyze the time complexity of Algorithm QUICKSORT depicted in Figure 6.11 which sorts an array a in index interval $[l, r]$ of size $n = r - l + 1$ in ascending order. The algorithm chooses some index p in this interval which determines a *pivot* value $a[p]$. It then calls a subalgorithm PARTITION that reorders $a[l \ldots r]$ such that, for some index m returned by PARTITION, $a[l \ldots m]$ holds all elements of a less than the pivot, $a[m]$ holds the pivot, and $a[m+1 \ldots r]$ holds all elements of a greater than or equal the pivot. By calling itself recursively twice on the intervals $[l, m-1]$ and $[m+1, r]$, the reordered array is sorted.

We don't yet specify the exact choice of p, but demand that it can be performed in time $O(n)$,

	Cost
procedure QUICKSORT(a, l, r) ▷ $n = r - l + 1$	1
if $l < r$ **then**	$O(n)$
choose $p \in [l, r]$	$\Theta(n)$
$m \leftarrow$ PARTITION(a, l, r, p) ▷ $i = m - l$	$1 + T(i)$
QUICKSORT$(a, l, m-1)$	$1 + T(n-i-1)$
QUICKSORT$(a, m+1, r)$	
end if	
end procedure	

Figure 6.11.: Quicksort

because this time is contained in the time $\Theta(n)$ that PARTITION apparently needs to subsequently reshuffle the array. The time complexity of QUICKSORT is thus determined by the recurrence

$$T(n) = T(i) + T(n - i - 1) + \Theta(n)$$

where $i := m - l$ is the size of the interval $[l, m-1]$ and $n - i - 1$ is the size of the interval $[m+1, n]$; consequently $0 \leq i \leq n - 1$.

Worst and Best Case Time Complexity If p is chosen such that $i = 0$ or $i = n - 1$ (i.e., one of the partition intervals is empty), we get the recurrence

$$T(n) = T(0) + T(n-1) + \Theta(n) = \Theta(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n)$$

which can be apparently solved as

$$T(n) = \sum_{i=0}^{n-1} \Theta(i) = \Theta(n^2)$$

In this case, the asymptotic time complexity of QUICKSORT is the same as that of INSERTIONSORT, i.e., it is quadratic in the size of (the range of) the array to be sorted. This case represents the worst-case time complexity of QUICKSORT (it corresponds to an unbalanced binary recursion tree where the left child of every node is a leaf and the longest path, the one from the root along all right children to the right-most leaf, has length n).

However, if p is chosen such that $i = \frac{n}{2}$ (i.e., both partition intervals have approximately the same size), we get the recurrence

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$$

We then can apply the second case of the master theorem for $a = b = 2$ to derive

$$T(n) = \Theta(n \cdot \log n)$$

i.e., the asymptotic time complexity of QuickSort is in this case the same as that of MergeSort. Since it can be shown that sorting an array of length n by comparing its elements needs at least time $\Theta(n \cdot \log n)$, this represents the best case (it corresponds to a balanced binary recursion tree where all paths have length $\log_2 n$).

A more detailed complexity analysis reveals that the constant factor attached to $n \cdot \log n$ is for QuickSort smaller than for MergeSort; in practice QuickSort thus outperforms MergeSort in the best case. Naturally the question arises whether the average-case time complexity of QuickSort is closer to the worst case or closer to the best case.

Average Case Time Complexity Let us for the moment assume that for i all n values $0, \ldots, n-1$ are equally likely (we will discuss this assumption later). Then the average-case time complexity is determined by the recurrence

$$\begin{aligned}
T(n) &= \frac{1}{n} \cdot \sum_{i=0}^{n-1} (T(i) + T(n-i-1) + \Theta(n)) \\
&= \frac{1}{n} \cdot \Big(\sum_{i=0}^{n-1} T(i) + T(n-i-1)\Big) + \Theta(n) \\
&= \frac{1}{n} \cdot \Big(\sum_{i=0}^{n-1} T(i) + \sum_{i=0}^{n-1} T(n-i-1)\Big) + \Theta(n) \\
&= \frac{1}{n} \cdot \Big(\sum_{i=0}^{n-1} T(i) + \sum_{i=0}^{n-1} T(i)\Big) + \Theta(n) \\
&= \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i) + \Theta(n)
\end{aligned}$$

This recurrence is particularly nasty because it also contains a sum. To derive a guess for its solution, we consider the more special form

$$T'(n) = \frac{2}{n} \cdot \sum_{i=1}^{n-1} T'(i) + n$$

which also avoids the index $i = 0$ because of step (*) shown below. To get rid of the sum, we first derive the two equalities

$$\begin{aligned}
n \cdot T'(n) &= 2 \cdot \sum_{i=1}^{n-1} T'(i) + n^2 \\
(n-1) \cdot T'(n-1) &= 2 \cdot \sum_{i=1}^{n-2} T'(i) + (n-1)^2
\end{aligned}$$

and then, by computing the difference of both sides, the equality

$$n \cdot T'(n) - (n-1) \cdot T'(n-1) = 2 \cdot T'(n-1) + 2n - 1$$

By simplification, we get the recurrence

$$n \cdot T'(n) = (n+1) \cdot T'(n-1) + 2n - 1$$

We divide by $n \cdot (n+1)$ and get the form

$$\frac{T'(n)}{n+1} = \frac{T'(n-1)}{n} + \frac{2n-1}{n \cdot (n+1)}$$

in which the terms involving T' have the same "shape" on the left and on the right hand side, which will help us to solve this recurrence. For this purpose, we compute

$$\begin{aligned}
\sum_{i=1}^{n} \frac{T'(i)}{i+1} &= \sum_{i=1}^{n} \Big(\frac{T'(i-1)}{i} + \frac{2i-1}{i \cdot (i+1)}\Big) \\
&= \sum_{i=1}^{n} \frac{T'(i-1)}{i} + \sum_{i=1}^{n} \frac{2i-1}{i \cdot (i+1)} \\
&\overset{(*)}{=} \sum_{i=0}^{n-1} \frac{T'(i)}{i+1} + \sum_{i=1}^{n} \frac{2i-1}{i \cdot (i+1)}
\end{aligned}$$

where in step (*) we shift in the first sum the index by one such that on the left side and on the right side now the same summand $\frac{T'(i)}{i+1}$ appears. By subtracting $\sum_{i=1}^{n-1} \frac{T'(i)}{i+1}$ from both sides we thus get

$$\frac{T'(n)}{n+1} = \frac{T'(0)}{1} + \sum_{i=1}^{n} \frac{2i-1}{i \cdot (i+1)}$$

respectively

$$T'(n) = (n+1) \cdot \left(T'(0) + \sum_{i=1}^{n} \frac{2i-1}{i \cdot (i+1)}\right)$$

By dropping the multiplicative and additive constants, we may derive the asymptotic bound

$$T'(n) = O\big(n \cdot \sum_{i=1}^{n} \frac{1}{i}\big) = O(n \cdot H_n) = O(n \cdot \log n)$$

where H_n denotes the already introduced n-th harmonic number which can be approximated as $\ln(n) + \frac{1}{2n} = O(\log n)$.

Actually, we could have got the same result by solving (a simplified form of) the recurrence directly with a computer algebra system, e.g. Maple:

```
> rsolve({T(0)=1,T(n)=(n+1)/n*T(n-1)+1},T(n));
                          (n + 1) (Psi(n + 2) + gamma)
```

Here the function `Psi(n)` denotes $\Psi(n) = H_{n-1} - \gamma$, i.e. the result is essentially $(n+1) \cdot H_{n+1} = O(n \cdot H_n)$. The corresponding result derived with Mathematica is

```
In[1]:= RSolve[{T[0]==1,T[n]==(n+1)/n*T[n-1]+1},T[n],n]
Out[1]= {{T[n] -> EulerGamma + EulerGamma n +
                  PolyGamma[0, 2 + n] + n PolyGamma[0, 2 + n]}}
```

where $\Psi(n)$ is denoted by `PolyGamma[0,n]`.

We thus estimate that the solution of the recurrence

$$T(n) = \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i) + \Theta(n)$$

is in $O(n \cdot \log n)$, i.e., that under the assumption that all values of i are equally likely, that the time complexity of QuickSort is in the average case the same as in the best case. Before discussing this assumption, we are going to verify our estimation in detail.

Proof We show that the solution of the recurrence

$$T(n) = \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i) + \Theta(n)$$

is in $O(n \cdot \log n)$. From the recurrence, we know there exist some $N \in \mathbb{N}$ and some $c \in \mathbb{R}_{>0}$ such that for all $n \geq N'$

$$T(n) \leq \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i) + c \cdot n$$

From the definition of $O(n \cdot \log n)$, we have to show, for some $N' \in \mathbb{N}$ and $c' \in \mathbb{R}_{>0}$, that for all $n \geq N'$

$$T(n) \leq c' \cdot n \cdot \log_2 n$$

To ensure that $N' \geq N$ and $\log_2 N' \geq 1$, we choose

$$N' := \max\{2, N\}.$$

Since later in the induction step for value n, we will apply the induction hypothesis to values $2, \ldots, n-1$, we have to show the induction base for all values $n \in \{2, \ldots, N'\}$ (rather than for $n = N'$ alone). Since $n \geq 2$ and $\log_2 n \geq 1$, to ensure

$$T(n) \leq c' \cdot n \cdot \log_2 n$$

it suffices to choose c' such that $c' \geq T(n)$. Since $T(2) \leq \ldots \leq T(N')$, it thus suffices to choose

$$c' := \max\{\text{␣}, T(N')\}$$

where ␣ is a quantity that will be defined in the remainder of the proof.

In the induction step, we take arbitrary $n > N'$ and assume that for all i with $2 \leq i < n$

$$T(i) \leq c' \cdot i \cdot \log_2 i$$

Our goal is to show that

$$T(n) \leq c' \cdot n \cdot \log_2 n$$

We know

$$T(n) \leq \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i) + c \cdot n$$

$$
\begin{aligned}
&= \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + \frac{2}{n} \cdot \big(T(0) + T(1)\big) + c \cdot n \\
&\leq \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + T(0) + T(1) + c \cdot n \\
&\leq \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + (T(0) + T(1) + c) \cdot n \\
&= \frac{2}{n} \cdot \sum_{i=2}^{n-1} T(i) + c'' \cdot n
\end{aligned}
$$

where $c'' := T(0) + T(1) + c$ and the last two inequalities hold because $n > N' \geq 2$. Since $N' \geq 2$, we can apply the induction hypothesis to all summands of $\sum_{i=2}^{n-1} T(i)$ and thus have

$$
\begin{aligned}
T(n) &\leq \frac{2}{n} \cdot \big(\sum_{i=2}^{n-1} c' \cdot i \cdot \log_2 i\big) + c'' \cdot n \\
&= \frac{2 \cdot c'}{n \cdot \ln 2} \cdot \big(\sum_{i=2}^{n-1} i \cdot \ln i\big) + c'' \cdot n \\
&\overset{(1)}{\leq} \frac{2 \cdot c'}{n \cdot \ln 2} \cdot \big(\int_2^n i \cdot \ln i \; di\big) + c'' \cdot n \\
&\overset{(2)}{=} \frac{2 \cdot c'}{n \cdot \ln 2} \cdot \big(\frac{i^2 \cdot \ln i}{2} - \frac{i^2}{4}\big)\Big|_2^n + c'' \cdot n \\
&= \frac{c'}{n \cdot 2 \cdot \ln 2} \cdot \big(2 \cdot i^2 \cdot \ln(i) - i^2\big)\Big|_2^n + c'' \cdot n \\
&= \frac{c'}{n \cdot 2 \cdot \ln 2} \cdot (2 \cdot n^2 \cdot \ln(n) - n^2 - 8 \cdot \ln 2 + 4) + c'' \cdot n \\
&\leq \frac{c'}{n \cdot 2 \cdot \ln 2} \cdot (2 \cdot n^2 \cdot \ln(n) - n^2) + c'' \cdot n \\
&= \frac{c' \cdot n}{2 \cdot \ln 2} \cdot (2 \cdot \ln(n) - 1) + c'' \cdot n
\end{aligned}
$$

Here we use in (1) the fact that the area under the continues curve $i \cdot \ln i$ in range $2 \leq i \leq n$ overapproximates the corresponding discrete sum in range $2 \leq i \leq n-1$; in (2) we use $\int i \cdot \ln i \; di = \frac{i^2}{4} \cdot (2 \cdot \ln(i) - 1)$ (please check by derivation). We now have to ensure that

$$
\begin{aligned}
\frac{c' \cdot n}{2 \cdot \ln 2} \cdot (2 \cdot \ln(n) - 1) + c'' \cdot n \leq c' \cdot n \cdot \log_2 n \;&\Leftrightarrow \\
\frac{c'}{2 \cdot \ln 2} \cdot (2 \cdot \ln(n) - 1) + c'' \leq c' \cdot \frac{\ln(n)}{\ln 2} \;&\Leftrightarrow \\
c' \cdot (2 \cdot \ln(n) - 1) + c'' \cdot 2 \cdot \ln 2 \leq c' \cdot 2 \cdot \ln(n) \;&\Leftrightarrow
\end{aligned}
$$

$$
\begin{aligned}
c'' \cdot 2 \cdot \ln 2 \leq c' \cdot (2 \cdot \ln(n) - 2 \cdot \ln(n) + 1) \;&\Leftrightarrow \\
c'' \cdot 2 \cdot \ln 2 \leq c' \;&
\end{aligned}
$$

We thus take $c' := \max\{c'' \cdot 2 \cdot \ln 2, T(N')\}$ and are done. □

Ensuring the Average Case Time Complexity As shown above, if we assume that all values of $i = m - l$ are equally likely, the asymptotic time complexity of QuickSort is in the average case the same as in the best case. This assumption is satisfied if and only if the choice

choose $p \in [l, r]$

determines a pivot element $a[p]$ that is equally likely to be the element at any of the positions $l, \ldots, r$ in the sorted array, i.e., if pivot elements are evenly distributed. We thus must discuss the implementation of the choice in some more detail.

The simplest implementation is the choice of a fixed index in interval $[l, r]$, e.g.,

$p \leftarrow r$

However, we then get evenly distributed pivot elements only if all $n!$ permutations of the array occur with equal probability as inputs. Unfortunately, it is in practice typically hard to estimate how inputs are distributed; it might, e.g., be frequently the case that QuickSort is called with arrays that are already sorted, either in the correct (ascending) order or exactly in the opposite (the descending) order. Both situations yield the extreme cases $i = n - 1$ respectively $i = 0$ which result in complexity $O(n^2)$.

To make all input permutations evenly likely, we might have the idea to permute the input array in a *random* fashion before we pass it to QuickSort. For instance, we may assume the availability of a a random number generator Random(a, b) that, for $a \leq b$, returns an integer in interval $[a, b]$, with each integer being returned with equal probability. We may then devise the following procedure Randomize:

```
procedure Randomize(a)
    n ← length(a)
    for i from 0 to n − 1 do
        r ← Random(i, n − 1)
        b ← a[i]; a[i] ← a[r]; a[r] ← b
    end for
end procedure
```

In this algorithm, before iteration i of the loop a random permutation of length i has been constructed whose probability is $\frac{(n-i)!}{n!}$. When the loop terminates with $i = n$, thus a random permutation of length n has been constructed whose probability is $\frac{1}{n!}$. If we call Randomize before calling QuickSort, then the implementation of the choice by $p \leftarrow r$ cannot cause any harm; any distribution of inputs becomes by randomization a uniform distribution for which the average case time complexity $O(n \cdot \log n)$ holds.

However, for QuickSort also a simpler and computationally much less costly strategy for randomization is possible. Rather than randomly permuting the input, we may randomly permute the choice by computing

$$\boxed{p \leftarrow \textsc{Random}(l, r)}$$

Since p is a random value with equal probability in $[l, r]$, also i becomes a random value with equal probability in $[l, r]$ and thus the average case time complexity $O(n \cdot \log n)$ is ensured.

Randomization is a powerful technique in the toolbox of algorithm design. Whenever in an algorithm some "choice" has to be made, making a random choice (rather than making an arbitrary fixed choice) may give us an algorithm whose average case complexity is independent of any assumption on the distribution of inputs; in particular, we may avoid worst case behaviors that might otherwise frequently happen. With the help of randomization, we may also develop *probabilistic* algorithms that may with a certain probability fail to return a result but that are in average faster than any deterministic counterparts; in some areas (such as distributed systems) probabilistic algorithms are even the only means of solving a problem.

6.5. Amortized Analysis

amortisierte Analyse

In *amortized analysis*, the goal is to determine the worst-case time complexity $T(n)$ of a *sequence* of n operations, such that the average-case time complexity of a single operation, also called the amortized cost, can be determined as $\frac{T(n)}{n}$. The idea is that, even if few of these operations have a big complexity, by the many other operations that have small complexity, the average complexity of a single operation may be also small. However, in contrast to the analysis of average-case time complexity shown in the previous sections, we are actually considering the worst case for the whole sequence of operations, i.e., we are computing the average performance of each operation in the worst case. This kind of analysis is typically applied to operations that manipulate a certain data structure (e.g., a sequence of method calls on an object). We will investigate for this purpose two techniques, *aggregate analysis* and the *potential method*.

Aggregate Analysis Let us consider the data structure *stack* with the following operations:

- Push(s, x): push an element x on stack s.
- Pop(s): pop an element from the top of the non-empty stack s.
- MultiPop(s, k): pop k elements from the stack s (if s has $l < k$ elements, then only l elements are popped).

Operations Push and Pop can be performed in time $O(1)$, while the complexity of MultiPop(s, k) is in $O(\min\{l, k\})$ where l is the number of elements on s. Our goal is to analyze the worst-case time complexity of an arbitrary sequence of n of these operations performed on a stack s which is initially empty. Since the size of the stack is at most n and the most time consuming operation is MultiPop with complexity $O(n)$, a sequence of n operations can be performed in time $O(n^2)$. However, while this analysis gives a correct upper bound, this bound is not tight; we show this below by a more detailed analysis in which *aggregate* the costs of all operations in the sequence.

Let us assume that in the sequence of n operations there occur k MultiPop operations. Consequently, the sequence starts with n_0 other operations before MultiPop is called for the first time, then there follow n_1 operations before MultiPop is called for the second time, and so on, until there are n_{k-1} operations before MultiPop is called for the last time; the sequence is then trailed by n_k other operations. In total we thus have

$$n = k + \sum_{i=0}^{k} n_i$$

operations. If we denote by p_i the number of elements actually popped from the stack in call i of MultiPop, the total cost of the sequence is

$$T(n) = \sum_{i=0}^{k-1} O(p_i) + (n-k) \cdot O(1) = O\Big(\sum_{i=0}^{k-1} p_i\Big) + O(n)$$
$$\overset{(1)}{=} O\Big(\sum_{i=0}^{k-1} n_i\Big) + O(n) \overset{(2)}{=} O(n) + O(n) = O(n)$$

Equality (1) is a consequence of $\sum_{i=0}^{k-1} p_i \leq \sum_{i=0}^{k-1} n_i$, which holds, because the total number of elements popped from the stack by the k MultiPop operations is bound by the total number of previously occuring Push operations. Equality (2) follows from $\sum_{i=0}^{k-1} n_i \leq n$, which holds, because the number of Push and Pop operations is bound by the number of all operations. A sequence of n operations can thus be actually performed in time $O(n)$, i.e., in linear (not

quadratic) time. The amortized cost of a single operation is thus $\frac{O(n)}{n} = O(1)$, i.e., it is *constant*. The costs for the linear time MultiPop operations have thus been outweighted be the average costs of the constant time Push and Pop operations.

The Potential Method We are now going to repeat the analysis of the sequence of *stack* operations in a more general framework that is useful for the analysis of more complex cases. The framework assigns to every operation i in a sequence of n operations two kinds of costs:

- the actual cost c_i, and
- the amortized cost $\hat{c}_i$

such that the sum of the amortized costs of all operations is an upper bound for the sum of the actual costs, i.e.,

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

For a single operation i, however, it may be the case that $\hat{c}_i < c_i$; the operation then uses up $c_i - \hat{c}_i$ "credit" that has been built up by the other operations. On the contrary, if $\hat{c}_i > c_i$, then operation i saves $\hat{c}_i - c_i$ credit that may be used up by other operations.

We will calculate the amortized costs with the help of a *potential function* $\Phi(s)$ that maps the data structure s manipulated by the operations to a real number, the *potential* of s (essentially representing the accumulated credit so far). Let s_0 denote the initial value of the data structure and s_i denote its value after operation i. Then we define for some constant $C > 0$ (a "scaling factor" chosen to simplify the later analysis)

$$\Phi(s_i) := (1/C) \cdot \sum_{j=0}^{i} (\hat{c}_j - c_j)$$

such that we have

$$\hat{c}_i - c_i = C \cdot \big(\Phi(s_i) - \Phi(s_{i-1})\big)$$

i.e., it is essentially the *difference* between the value of Φ after execution of operation i and the value of Φ before the execution which represents the credit saved/used by the operation. We can then determine the total sum of the amortized costs as

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} \Big(c_i + C \cdot \big(\Phi(s_i) - \Phi(s_{i-1})\big)\Big) \\ &= \sum_{i=1}^{n} c_i + C \cdot \big(\Phi(s_n) - \Phi(s_0)\big)\end{aligned}$$

where the last equality holds because in the sum, except for the first and the last term, subsequent terms $\Phi(s_i)$ cancel (the sum *telescopes*). To ensure $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$, i.e, that the amortized costs are an upper bound for the actual costs, it thus suffices to ensure

$$\Phi(s_n) \geq \Phi(s_0)$$

i.e., that the potential of the data structure after the execution of the operations is not smaller than its initial potential.

In the *stack* example, we define $\Phi(s)$ as the number of elements in stack s. Then clearly

$$\Phi(s_n) \geq 0 = \Phi(s_0)$$

i.e., the amortized costs described by the potential function represent an upper bound for the actual costs. Furthermore, we choose $C > 0$ such that it is an upper bound for the execution time of Push and Pop and that $C \cdot k'$ is an upper bound for the execution time of MultiPop(s, k) where $k' = \min\{k, m\}$ and m is the number of elements in s (it can be shown that such a C exists because the complexities of the operations are $O(1)$ respectively $O(k')$). We can then determine the amortized cost

$$\hat{c}_i = c_i + C \cdot \big(\Phi(s_i) - \Phi(s_{i-1})\big)$$

of each operation:

- Push(s, x):
$$\hat{c}_i \leq C + C \cdot \big((m+1) - m\big) = C + C = 2 \cdot C$$
where m is the number of elements on s.
- Pop(s):
$$\hat{c}_i \leq C + C \cdot \big((m-1) - (m)\big) = C - C = 0$$
where $m > 0$ is the number of elements on s.
- MultiPop(s, k):
$$\hat{c}_i \leq C \cdot k' + C \cdot \big((m - k') - m\big) = C \cdot k' - C \cdot k' = 0$$
where $k' := \min\{k, m\}$ and m is the number of elements in s.

Unlike in aggregate analysis, we can thus analyze with the potential method the amortized

cost of each operation in turn. Since each operation has amortized cost $O(1)$, a sequence of n operations has worst-case time complexity $O(n)$.

Dynamic Tables We are now going to consider an operation $\textsc{Insert}(t, x)$ that inserts a value x into a table t for which a fixed amount of space is allocated. If this space gets exhausted, more space is allocated and the elements are copied from the old space to the new one; we say that the table is *expanded*. If n elements are to be copied, the worst case complexity of Insert is thus $O(n)$. Most of the time, however, there is still free space left to accommodate x; if this is indeed the case, then Insert can be performed in time $O(1)$. Our goal is to determine the time complexity of a sequence of n Insert operations starting with an empty table (with no memory allocated yet). Clearly, since the time for a single insert is bound by $O(n)$, a bound for n operations is $O(n^2)$. However, this bound is (hopefully) not tight, thus we are interested to find a better bound.

For a more detailed analysis, we have to know how much a table of size m is expanded if Insert detects that it has become full. Clearly it is a bad idea to just expand the table to size $m + 1$, because then immediately the next call of Insert will require another expansion. As we will now show, it is also not much of an improvement to expand to size $m + c$ for some constant c. By this strategy, for every c calls of Insert an expansion of the table is triggered, where the i-th expansion requires $(i - 1) \cdot c$ copy operations. If we have n calls of `Insert`, there occur $\lfloor \frac{n}{c} \rfloor$ expansions and $n - \lfloor \frac{n}{c} \rfloor$ calls that do not trigger an expansion and can be thus performed in constant time; the total time for n Insert operations is thus

$$\begin{aligned} T(n) &= \sum_{i=1}^{\lceil \frac{n}{c} \rceil} O\big((i-1) \cdot c\big) + \big(n - \lceil \frac{n}{c} \rceil\big) \cdot O(1) \\ &= O\big(\sum_{i=1}^{\lceil \frac{n}{c} \rceil} i\big) + O(n) \\ &= O(n^2) + O(n) = O(n^2) \end{aligned}$$

In other words, the asymptotic complexity is still quadratic.

A better strategy is to allocate $\lceil m \cdot c \rceil$ cells for some constant $c > 1$. A popular choice is to use $c := 2$, i.e., to double the size of the space from m to $2 \cdot m$, because then after the allocation still at least 50% of the space is used, so not more space is unfilled than filled. Then in a sequence of Insert operations numbered $i \in \{1, \ldots, n\}$, the ones with number $i \in \{2 = 1 + 1, 3 = 2 + 1, 5 = 4 + 1, 9 = 8 + 1, \ldots, 2^{\lfloor \log_2 n \rfloor} + 1\}$ lead to a costly allocation; more specifically, in operation $2^i + 1$ we have to allocate a new table and copy 2^i elements from the old one. Using aggregate analysis, we can thus determine the total cost for n operations as

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lceil \log_2 n \rceil + 1} O(2^i) + (n - \lceil \log_2 n \rceil - 1) \cdot O(1) \\ &= O\big(\sum_{i=0}^{\lceil \log_2 n \rceil + 1} 2^i\big) + O(n) \\ &= O\big(\sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i\big) + O(n) \\ &= O(2n) + O(n) = O(n) \end{aligned}$$

The amortized cost of a single Insert is thus $\frac{O(n)}{n} = O(1)$, i.e., every Insert operation is in average performed in constant time.

A more detailed analysis becomes possible by the potential method. Let $num(t)$ denote the number of elements in table t and $size(t)$ denote the number of elements that can be stored in the table (including the empty slots); by the strategy to double the space when the table gets full, we always have $num(t) \geq \frac{size(t)}{2}$. Initially $num(t) = size(t) = 0$; when a new element is inserted than $num(t)$ is incremented by 1. If more memory is to be allocated, then $size(t)$ is multiplied by 2. Based on $num(t)$ and $size(t)$, we can define the potential function

$$\Phi(t) := 2 \cdot num(t) - size(t)$$

for which, since $num(t) \geq \frac{size(t)}{2}$, we know

$$\Phi(t) \geq 0$$

The idea of this definition is as follows: immediately after an expansion of table t_{i-1} to table t_i, we have $num(t_i) = \frac{size(t_i)}{2}$ and thus $\Phi(t) = 0$. Whenever a new element is inserted to non-full t_{i-1}, we have $\Phi(t_i) = 2 + \Phi(t_{i-1})$. When t_{i-1} becomes full, we have $num(t_{i-1}) = size(t_{i-1})$ and thus $\Phi(t_{i-1}) = size(t_{i-1})$. The $size(t_{i-1})$ copy operations of the subsequent expansion can be thus performed at the expense of the credit accumulated since the last expansion.

Based on this definition of Φ, will now analyze in detail the amortized cost

$$\hat{c_i} = c_i + C \cdot \big(\Phi(t_i) - \Phi(t_{i-1})\big)$$

of Insert operation number i which applied to table t_{i-1} yields table t_i. Here we choose C such that it is an upper bound for the time of Insert when t_{i-1} is not expanded (which can be

performed in time $O(1)$) and $C \cdot n$ is an upper bound for the time when t_{i-1} with n elements has to be expanded (which can be performed in time $O(n)$).

- If INSERT does not trigger an expansion, then we have $num(t_i) = num(t_{i-1}) + 1$ and $size(t_i) = size(t_{i-1})$ and thus:

$$\begin{aligned}\hat{c}_i &\leq C \cdot 1 + C \cdot \big(2 \cdot num(t_i) - size(t_i) - (2 \cdot num(t_{i-1}) - size(t_{i-1}))\big) \\ &= C \cdot 1 + C \cdot \big(2 \cdot (num(t_{i-1}) + 1) - size(t_{i-1}) - (2 \cdot num(t_{i-1}) - size(t_{i-1}))\big) \\ &= C \cdot 1 + C \cdot 2 = 3C\end{aligned}$$

  where $C \cdot 1$ denotes an upper bound for the cost for the insertion of the new element.

- If INSERT triggers an expansion, then we have $num(t_i) = num(t_{i-1}) + 1$ and $size(t_{i-1}) = num(t_{i-1})$ and $size(t_i) = 2 \cdot size(t_{i-1}) = 2 \cdot num(t_{i-1})$ and thus:

$$\begin{aligned}\hat{c}_i &\leq C \cdot num(t_{i-1}) + C \cdot \big(2 \cdot num(t_i) - size(t_i) - (2 \cdot num(t_{i-1}) - size(t_{i-1}))\big) \\ &= C \cdot num(t_{i-1}) + C \cdot \big(2 \cdot (num(t_{i-1}) + 1) - 2 \cdot num(t_{i-1}) - (2 \cdot num(t_{i-1}) - num(t_{i-1}))\big) \\ &= C \cdot num(t_{i-1}) + C \cdot \big(2 - num(t_{i-1})\big) = 2C\end{aligned}$$

  where $C \cdot num(t_{i-1})$ denotes an upper bound for the cost for copying the $num(t_{i-1})$ elements from t_{i-1} to t_i.

In both cases, we thus have $\hat{c}_i = O(1)$, i.e., we can expect to perform INSERT in constant time.

Chapter 7.

Limits of Feasibility

In Chapter 5 we have investigated different classes of computational complexity; in this chapter we will focus on the difference between the class $\mathcal{P}$ of those problems whose computations are considered as *feasible*, because they can be performed in polynomial time by a deterministic Turing machine, and the complementary class of those problems that are considered as *infeasible*. We start with a discussion on how the complexity of Turing machine computations is related to that of other Turing complete computational models. Then we elaborate the relationship of different classes of problems whose solutions have the same complexity; as it turns out, an important open problem is whether $\mathcal{P} = \mathcal{NP}$, i.e., whether the application of nondeterminism is able to speedup a computation from non-polynomial to polynomial time complexity or not. The core answer to this question lies with the important class of $\mathcal{NP}$*-complete* problems which we will investigate in some more detail.

7.1. Complexity Relationships Among Models

All Turing complete computational models have the same expressive power in that they can compute the same class of functions. However, the "same" computations performed in different models may take a different amount of time respectively space, because

1. each model may have a different notion of time and space consumption (the functions $t(i)$ and $s(i)$ in Definition 44);

2. the encoding of an input in each model may have different size (the function $|i|$ in Definition 44).

Consequently, the "same" computation may, when it is performed in different computational models, even lie in different complexity classes.

Example 34 Take the computation, for given $n \in \mathbb{N}$, of the value $r := 2^{2^n}$ by the algorithm

$r := 2;\ i := 0$
while $i < n$ **do**
    $r := r * r;\ i := i + 1$

We are going to investigate the complexity of this algorithm on a random access machine under different cost models.

In a random access machine, every cell of the input tape and every register may hold a natural number. For above algorithm the machine needs a constant number of registers, thus

$$S_1(n) = O(1)$$

The algorithm requires n iterations; since the computation of $i := i + 1$ takes a single instruction, the time of each iteration is asymptotically dominated by the multiplication $r * r$.

For determining the time complexity of multiplication, let us first assume a variant of a random access machine which supports an instruction `MUL (r)` which multiplies the accumulator with the content of register r. Since we now need only a single instruction to perform the multiplication, one iteration of the loop takes time $O(1)$. Since we have n iterations, the time complexity $T_1(n)$ of the algorithm in this variant of the random access machine model is

$$T_1(n) = \Theta(n)$$

If we return to our original model of a random access machine which only has the arithmetic instruction `ADD #c` that increments the accumulator by constant c, then the computation of $n + n$ takes time $\Theta(n)$ and the computation of $n * n$ takes time $\Theta(n^2)$. In our algorithm, the most time-consuming multiplication is the computation of $2^{2^{n-1}} * 2^{2^{n-1}}$ which takes time $\Theta(2^n)$. We can thus determine for the time complexity $T_2(n)$ of the algorithm the following bounds:

$$T_2(n) = \Omega(2^{2^n})$$
$$T_2(n) = O(n \cdot 2^{2^n})$$

Since T_1 and T_2 thus fall into widely different complexity classes, we might be inclined to reconsider how we measure time and space on a random access machine, i.e., what *cost model* we use. The first model imposed extremely high demands by requiring the multiplication of arbitrarily big numbers in a single step. The second model only had extremely low demand by requiring only the increment by a constant in a single step. Both cost models thus represent the

opposite ends of a spectrum both of which seem a bit "unrealistic". We are thus interested in determining what a "realistic" cost model for a random access machine might be.

First of all, we should find a realistic representation of number n; up to now we considered $|n| := 1$, i.e., an arbitrarily large number can be represented in a single tape cell respectively memory register. In reality, we use for the representation of natural numbers a positional number system with some base b; in such a system, a number $n > 0$ can be represented by $l(n) := 1 + \lfloor \log_b n \rfloor = \Theta(\log n)$ digits; since b is constant, a random access machine can perform arithmetic on individual digits in time $O(1)$. For measuring the space complexity of the computation, it is thus realistic to measure the digit length of every number stored in every cell of the input tape respectively in every register of the memory, i.e. $|n| = \Theta(\log n)$. Furthermore, in a positional system, the addition $n + n$ can be performed in time $\Theta(\log n)$ and the multiplication $n * n$ in time $O((\log n)^2)$ (a tight lower bound for the time complexity of multiplication is not known, it is clearly in $\Omega(\log n)$ and conjectured to be in $\Omega((\log n) \cdot (\log \log n))$), thus it is realistic to assign to every addition and every multiplication these costs.

We will now apply this "realistic" cost criterion to our computation. Since we have to store a fixed amount of numbers, of which the largest is 2^{2^n}, we get the "realistic" space complexity

$$S(n) = \Theta(\log 2^{2^n}) = \Theta(2^n)$$

The sequence of multiplication takes time

$$O((\log 2)^2 + (\log 4)^2 + \ldots + (\log 2^{2^{n-1}})^2) = O(1^2 + 2^2 + \ldots + (2^{n-1})^2) = O\Big(\sum_{i=0}^{n-1} (2^i)^2\Big)$$

It can be shown that

$$\sum_{i=0}^{n-1} (2^i)^2 = \frac{4^n - 1}{3}$$

We thus get the "realistic" time complexity

$$T(n) = O(4^n)$$

In comparison with the "unrealistic" models considered before we thus have

$$S_1(n) = o(S(n))$$
$$T_1(n) = o(T(n)) = o(T_2(n))$$

i.e., we need more space; the required time is between the previously established bounds. □

The "realistic" cost model presented in the example above is formalized as follows.

logarithmisches Kostenmodell

Definition 48 (Logarithmic Cost Model of RAM) The *logarithmic cost model* of a random access machine assigns to every number n stored on the input tape or in a register the size

$$|n| = \Theta(\log n)$$

The space complexity of a computation is the maximum, at any time during the computation, of the sums $\sum_n |n|$ of the sizes of all numbers stored in the random access machine. The time complexity of a computation is the sums of the execution times of all instructions executed by the machine, where every instruction which involves a number n on a tape or in a register is assigned time $|n|$ and every other instruction is assigned time 1.

We know already by Theorem 12 that every random access machine can be simulated by a Turing machine and vice versa. By careful analysis of the simulation, one can establish the following result (which we state without proof).

Theorem 50 (Complexity of Random Access Machine versus Turing Machine)

- Every computation of a Turing machine with time $O(T(n))$ can be simulated by a random access machine with time $O(T(n) \cdot \log T(n))$ (under the logarithmic cost model).
- Every computation of a random access machine with time $O(T(n))$ (under the logarithmic cost model) can be simulated by a Turing machine with time $O(T(n)^4)$.

Under the logarithmic cost model, the execution times of random access machines and Turing machines thus only differ by a *polynomial transformation*, i.e., if the time complexity of a computation in one model is polynomial, it remains also polynomial in the other model; in particular, by the simulation we do not cross the boundary between polynomial and exponential complexity (this would not be true, if we would use the cost model of the derivation of time $T_1(n) = O(n)$ in the example above; a Turing machines needs already $\Omega(2^n)$ time to write the result 2^{2^n} to the output tape).

Similar kinds of analysis have been performed for other Turing complete computational models. It has turned out that for a "realistic" cost assumption, the simulation of such a

model by a Turing machine and vice versa does not cost more than a polynomial slowdown, i.e, if a computation is polynomial in one model it also remains polynomial in the other one; furthermore, the simulation does not cost more than constant overhead in space. This has led to the following thesis that was formulated by Cees F. Slot and Peter van Emde Boas in 1984.

Thesis 2 (Invariance Thesis) "Reasonable" machines (i.e., Turing complete computational models) can simulate each other within a polynomially bounded overhead in time and a constant-factor overhead in space.

Of course, since the notion "reasonable" is an intuitive one, this is only an unprovable thesis, not a provable theorem. Furthermore, it is possible that quantum computation disproves the thesis (quantum algorithms that can solve some problems in polynomial time for which no classical polynomial time algorithms are known). Nevertheless, there seems to be some kind of fundamental border between computations that take polynomial time and computations that take more (in particular exponential) time. The relationship between these classes of computations will be investigated in the following sections.

7.2. Problem Complexity

We are now going to investigate classes of problems whose solution by a Turing machine have same asymptotic complexity. In this context, we will also consider nondeterministic Turing machines which in each step can "fork" their computations into multiple "branches" that are simultaneously investigated. Therefore we need an appropriate notion of resource consumption for nondeterministic Turing machines.

Definition 49 (Resource Consumption of Non-Deterministic Turing Machine) The time consumption $t(i)$ respectively space consumption $s(i)$ of a nondeterministic Turing machine M is the maximum number of configurations respectively maximum distance from the beginning of the tape that any run of M for input i has.

This notion of resource consumption is based on the intuition that is visualized in Figure 7.1, namely that a nondeterministic Turing machine can "simultaneously" investigate multiple

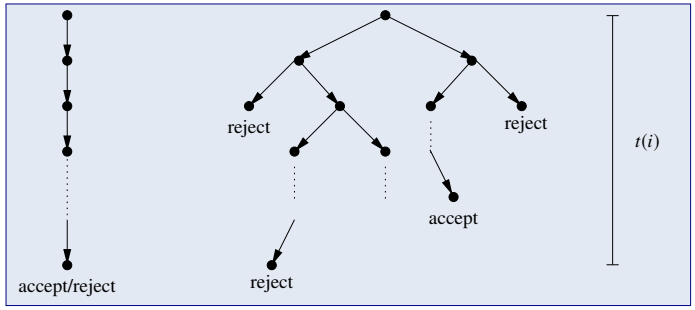


Figure 7.1.: Deterministic versus Nondeterministic Time Consumption

branches. However, one may then question the decision to consider the *maximum* time of *all* branches rather than the *minimum* time of the *accepting* ones. The rationale for our definition is that we are less interested in the execution times of an individual non-deterministic Turing machine but more in the execution times of very large classes of Turing machines such as the class $\mathcal{NP}$ of non-deterministic Turing machines that run in at most polynomial time. One can show that this class is the same under either of the two possibilities to define the execution time of a nondeterministic Turing machine. Since the difference does therefore not really matter and the first definition is easier to handle (because it just depends on upper bounds for all branches), it is the one that is preferred.

Based on the definitions of $t(i)$ respectively $s(i)$, the worst-case space complexity $T(n)$ and the worst-case space complexity $S(n)$ of a nondeterministic Turing machine are defined exactly as in Definition 44 for a classical Turing machine.

Example 35 We want to solve the problem of *Hamiltonian Cycles*: given an undirected graph G with n nodes, does there exist a cyclic path in G that visits every node exactly once?

A nondeterministic Turing machine M can solve this problem in the following way:

1. M writes non-deterministically n numbers in the range $1 \ldots n$ to the tape.
2. M checks whether the n numbers represent a Hamiltonian cycle in G.

We assume that G is represented on the input tape of M by a sequence of n^2 bits that describe, for every combination of node i and node j, whether i is connected to j. For every input of length b, we thus have $n = O(\sqrt{b})$ and consequently $n = O(b)$, i.e., the number of nodes n in G is asymptotically bounded by the length b of the input of M. In the following analysis, we will thus simply use n rather than b as the size of the input.

Since number n can be written in time $O(n)$, the first step can be performed in time $O(n^2)$. In the second step, for every number i on the tape, M checks whether i has not yet appeared on the tape; if yes, M checks whether i and its predecessor on the tape represent adjacent (i.e., neighboring) nodes in the graph. If all numbers written to the tape pass this check and if also the nodes represented by the last number and by the first number are adjacent, then M accepts G.

The second step can be performed by a random access machine M' in polynomial time:

- M' can check whether a new number i has already appeared by checking the value of register $R[c + i]$ (for some appropriate offset c); if its value is 0, the number has not yet occurred; the value is then set to 1 before the check of the next number. In the logarithmic cost model, the time for the computation of $c + i$ is $O(\log n)$ as is also the time for the register access; the total time complexity for all checks is thus $O(n \cdot \log n)$.

- M' can check whether two graph nodes are adjacent by first translating the representation of the at most n^2 edges of G on the input tape into a $n \times n$ adjacency matrix stored in the memory and then performing at most n matrix lookups. The total costs are thus dominated by the matrix construction which, if node i is adjacent to node j, sets the value of register $R[d + i \cdot j]$ to 1 (for some appropriate offset d). The time for this operation is dominated by the computation of the product $i \cdot j$ which can be performed in time $O(n^2)$. The total time complexity for the matrix construction is thus in $O(n^4)$.

By Theorem 50, then also M can perform this step in polynomial time. Thus M can solve the whole problem in polynomial time.

On the other side, a deterministic Turing machine has to implement a search for possible cycles in the graph; the fastest solutions known have exponential time complexity. □

Above example demonstrates that nondeterministic Turing machines can solve a problem by first *guessing* a possible solution and then *checking* whether the guess is correct. If both the guessing and the checking phase can be performed in polynomial time, then the problem can be solved in polynomial time, because all possible guesses can be enumerated and simultaneously investigated in different execution branches. On the other side, a deterministic Turing machine can solve a problem only by *constructing* a solution, which is intuitively more difficult than guessing and checking and thus also takes more time. In the following, we will investigate the relationship between these two approaches in more detail.

We start with a definition which allows us to categorize problems into classes according to the resources needed for their solution.

Raumkomplexität
Zeitkomplexität

Definition 50 (Problem Complexity) A decidable problem P has (deterministic/nondeterministic) *time complexity* $T(n)$ respectively (deterministic/nondeterministic) *space complexity* $S(n)$ if there exists a (deterministic/nondeterministic) Turing machine M that decides P, such that, for every input w with length $n = |w|$, M terminates in time $O(T(n))$ respectively uses space $O(S(n))$.

We define $DTIME(T(n))$, $NTIME(T(n))$, $DSPACE(T(n))$, and $NSPACE(T(n))$

$$\begin{aligned} DTIME(T(n)) &:= \{P \mid P \text{ has deterministic time complexity } T(n)\} \\ NTIME(T(n)) &:= \{P \mid P \text{ has nondeterministic time complexity } T(n)\} \\ DSPACE(T(n)) &:= \{P \mid P \text{ has deterministic space complexity } T(n)\} \\ NSPACE(T(n)) &:= \{P \mid P \text{ has nondeterministic space complexity } T(n)\} \end{aligned}$$

as the classes of all decidable problems with (deterministic/nondeterministic) time complexity $T(n)$ respectively (deterministic/ nondeterministic) space complexity $P(n)$.

Based on this definition, we can construct a number of important complexity classes (there are many more classes, the *Complexity Zoo*[1] currently lists almost 500 of them).

Definition 51 (Problem Complexity Classes) We define $\mathcal{P}$, $\mathcal{NP}$, $PSPACE$, and $NSPACE$

$$\begin{aligned} \mathcal{P} &:= \bigcup_{i \in \mathbb{N}} DTIME(n^i) \\ \mathcal{NP} &:= \bigcup_{i \in \mathbb{N}} NTIME(n^i) \\ PSPACE &:= \bigcup_{i \in \mathbb{N}} DSPACE(n^i) \\ NSPACE &:= \bigcup_{i \in \mathbb{N}} NSPACE(n^i) \end{aligned}$$

as the classes of all decidable problems that can be solved by a (deterministic/nondeterministic) Turing machine in polynomial time/space complexity.

Furthermore, we define $EXPTIME$ and $NEXPTIME$

$$EXPTIME := \bigcup_{i \in \mathbb{N}} DTIME(2^{n^i})$$

[1] https://complexityzoo.uwaterloo.ca/Complexity_Zoo

$$NEXPTIME := \bigcup_{i \in \mathbb{N}} NTIME(2^{n^i})$$

as the classes of all decidable problems that can be solved by a (deterministic/nondeterministic) Turing machine with exponential time complexity (where the exponent can be a polynomial n^i).

The fundamental role of $\mathcal{P}$ is illustrated by the following thesis which was formulated by Alan Cobham in 1965.

Thesis 3 (Cobham's Thesis) A problem is *tractable*, i.e., can be feasibly decided, if it is in complexity class $\mathcal{P}$.

If we accept this thesis and also the invariance thesis, the class of tractable problems is the same for every computational model, i.e., "feasibly computable" means "feasibly computable on any computational device".

The relationship between $\mathcal{P}$ and $\mathcal{NP}$ is illustrated by the following notion and theorem.

Prüfer

Definition 52 (Verifier) A *verifier* V for a language L is a Turing machine such that

$$L = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some word } c\}$$

i.e., for every word in L, there is some extra input c which we call the *certificate*, such that V accepts the pair $\langle w, c \rangle$. A *polynomial time verifier* is a verifier that runs in polynomial time in the length of w.

Zertifikat

Polynomialzeit-Prüfer

The role of the certificate c is to "drive" the execution of V in the decision whether to accept input w; this point is illustrated in the proof of the following theorem.

Theorem 51 (Polynomial Time Verification) A language L is decided by some nondeterministic Turing machine in polynomial time (i.e., $L \in \mathcal{NP}$) if and only if there exists a polynomial time verifier for L.

Proof We sketch the proof of both directions of the theorem.

⇒ Let L be decided by a nondeterministic Turing machine M in polynomial time. We can construct a polynomial time verifier V for L as follows: V takes input $\langle w, c\rangle$ where c denotes a sequence of choices to be made at each step for the execution of a nondeterministic Turing machine. V simulates the execution of M on that branch described by c. If w is in L, then there exists a branch of M that accepts w in polynomial time and a corresponding certificate c; thus V accepts $\langle w, c\rangle$ in polynomial time.

⇐ Let V be the polynomial time verifier for V i.e., V terminates in polynomial time $T(n)$ for an input w of length n. We can construct a non-deterministic Turing machine M that decides L in polynomial time as follows: on input w, M non-deterministically generates a certificate c of length $T(n)$ (i.e., every possible certificate is generated in some branch of the non-deterministic execution) and then simulates the execution of V on $\langle w, c\rangle$. If there is some c such that V accepts $\langle w, c\rangle$ in polynomial time, then also M accepts w in polynomial time. □

Above theorem formalizes the principle that every problem solution that can be deterministically verified in polynomial time can be also non-determinstically constructed in polynomial time and vice versa.

There are various relationships of the complexity classes defined above that immediately follow from their definition.

Theorem 52 (Some Relationships Between Complexity Classes)

1. $\mathcal{P} \subseteq \mathcal{NP}$
2. $EXPTIME \subseteq NEXPTIME$
3. $\mathcal{P} \subseteq PSPACE$
4. $\mathcal{NP} \subseteq NSPACE$
5. $PSPACE \subseteq EXPTIME$

Proof Parts 1 and 2 immediately follow from the fact that every deterministic Turing machine is a special case of a nondeterministic Turing machine.

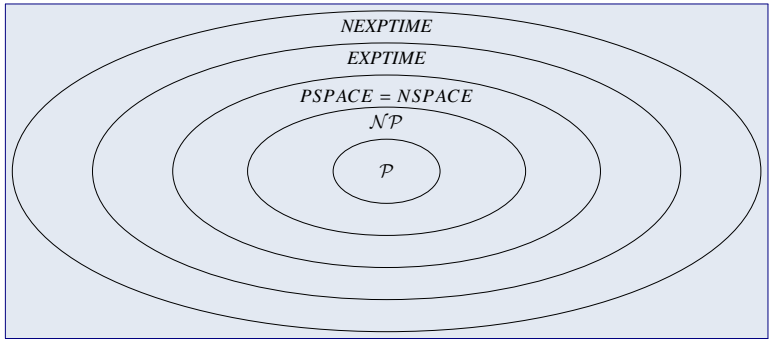


Figure 7.2.: Problem Complexity Classes

Parts 3 and 4 follow from the fact that every Turing machine, in order to consume one unit of space, has to write one cell to the tape, i.e., it consumes one unit of time. A Turing machines therefore cannot consume more memory than time.

Part 5 follows from the fact that, if the space of a Turing machine is constrained by a polynomial bound $O(n^i)$, then the number of configurations of the Turing machine is constrained by an exponential bound $O(2^{n^i})$; consequently the Turing machine cannot make more than exponentially many steps until it halts. □

However, there is also a non-trivial relationship that could be proved by Walter Savitch only in 1970 (we state the theorem without proof).

Theorem 53 (Savitch's Theorem) Every problem that is decidable by a nondeterministic Turing machine with space $O(S(n))$ is also decidable by a deterministic Turing machine with space $O(S(n)^2)$:

$$NSPACE(S(n)) \subseteq DSPACE(S(n)^2)$$

Consequently, $NSPACE = PSPACE$.

Combining the results of Theorem 52 and Theorem 53, we get the following core sequence of relationships illustrated in Figure 7.2:

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq PSPACE = NSPACE \subseteq EXPTIME \subseteq NEXPTIME$$

Now, the sad but true fact is that (almost) nothing else is known about the relationship of these classes. To be fair, it has been proved that

- $\mathcal{P} \neq EXPTIME$ and
- $\mathcal{NP} \neq NEXPTIME$

i.e., there exist problems that can be decided not faster than with exponential complexity, both deterministically and nondeterministically. Therefore at least one of the first three subset relationships and at least one of the last three subset relationships must be a proper one, but it is unknown which one it is.

In particular, it is not known whether

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$$

i.e., whether it is indeed easier to *guess* and *check* a solution to a problem than to *construct* the solution (i.e., $\mathcal{P} \subset \mathcal{NP}$) or whether both are actually equally complex (i.e., $\mathcal{P} = \mathcal{NP}$). While virtually all theoretical computer scientists believe $\mathcal{P} \subset \mathcal{NP}$, no one has ever come up with a correct proof, so the bets are still open. This is arguably the most important unsolved problem in theoretical computer science; it is one of the *Millennium Prize Problems* for whose first correct solution the Clay Mathematics Institute will pay US$1,000,000.

7.3. $\mathcal{NP}$-Complete Problems

In order to show $\mathcal{P} \neq \mathcal{NP}$, it suffices to find a single problem that is in $\mathcal{NP}$ but not in $\mathcal{P}$, i.e., a single problem that can be solved in polynomial time only by a *nondeterministic* Turing machine. The search for such a problem apparently focuses on the "hardest" (most time-consuming) problems in $\mathcal{NP}$. Therefore we need some way to compare problems according to how "hard" they can be solved.

Polynom-Zeit-reduzierbar

Definition 53 (Polynomial-Time-Reducibility) A decision problem $P \subseteq \Sigma^*$ is *polynomial time-reducible* to a decision problem $P' \subseteq \Gamma^*$ (written $P \leq_P P'$) if there is a function $f : \Sigma^* \to \Gamma^*$ such that for all $w \in \Sigma^*$

$$P(w) \Leftrightarrow P'(f(w))$$

and f can be computed by a deterministic Turing machine in polynomial time.

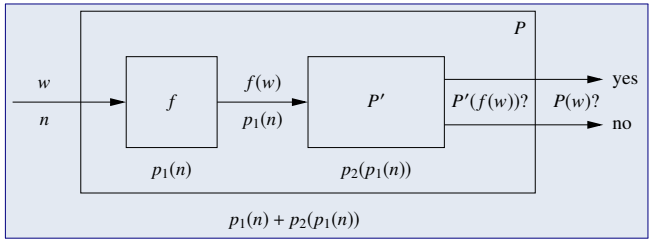


Figure 7.3.: Polynomial-Time-Reducibility

Intuitively, if $P \leq_P P'$, then a decision of P is "essentially" (up to a polynomial transformation) not more time-consuming than a decision of P'.

We then have the following result.

Theorem 54 (Polynomial-Time-Reducibility and $\mathcal{P}/\mathcal{NP}$) For all decision problems P and P' with $P \leq_P P'$, we have

$$P' \in \mathcal{P} \Rightarrow P \in \mathcal{P}$$
$$P' \in \mathcal{NP} \Rightarrow P \in \mathcal{NP}$$

In other words, if P is polynomial time-reducible to P' and P' can be decided in polynomial time by a (deterministic/nondeterministic) Turing machine, then also P can be decided in polynomial time by a (deterministic/nondeterministic) Turing machine.

Proof We only prove $P' \in \mathcal{P} \Rightarrow P \in \mathcal{P}$ (the proof of $P' \in \mathcal{NP} \Rightarrow P \in \mathcal{NP}$ is completely analogous). We assume $P \leq_P P'$ and $P' \in \mathcal{P}$ and show $P \in \mathcal{P}$ (see also Figure 7.3).

Since $P \leq_P P'$, there exists a function f such that, for every input w, $P(w) \Leftrightarrow P'(f(w))$ and $f(w)$ can be computed in time $p_1(n)$ for some polynomial p_1 and $n := |w|$. Since $f(w)$ can be computed in time $p_1(n)$, we have also $|f(w)| \leq p_1(n)$, because $p_1(n)$ includes the time that the Turing machine needs to write $f(w)$ to the tape and this time cannot be smaller than the length of $f(w)$. Furthermore, since $P' \in \mathcal{P}$, we can decide $P'(f(w))$ in time $p_2(|f(w)|) \leq p_2(p_1(n))$ for some polynomial p_2.

The total time for the decision of $P(w)$ is thus bound by the polynomial $p_1(n) + p_2(p_1(n))$ (the sum of the times for translating w to $f(w)$ and for deciding $P'(f(w))$, thus $P \in \mathcal{P}$. □

We now come to the main definition of this section.

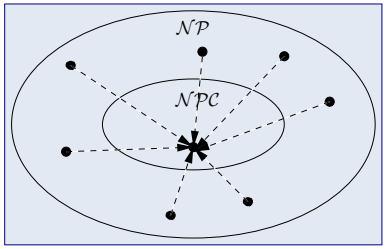


Figure 7.4.: $\mathcal{NP}$-Completeness

$\mathcal{NP}$-vollständig

Definition 54 ($\mathcal{NP}$-Completeness) A problem P' in complexity class $\mathcal{NP}$ is *NP-complete*, if for every problem $P \in \mathcal{NP}$ we have $P \leq_P P'$. Furthermore, we define

$$\mathcal{NPC} := \{P' \in \mathcal{NP} \mid P' \text{ is } \mathcal{NP}\text{-complete}\}$$

as the class of all $\mathcal{NP}$-complete problems.

Every problem in $\mathcal{NP}$ can thus be reduced in polynomial time to any problem in $\mathcal{NPC}$ (see Figure 7.4). The relevance of this class is illustrated by the following theorem.

Theorem 55 ($\mathcal{NPC}$ and $\mathcal{P} = \mathcal{NP}$) If $\mathcal{NPC} \cap \mathcal{P} = \emptyset$, then $\mathcal{P} \neq \mathcal{NP}$, and vice versa.

In other words, if no $\mathcal{NP}$-complete problem can be deterministically decided in polynomial time, then $\mathcal{P} \neq \mathcal{NP}$, and vice versa.

Proof To show the $\Rightarrow$ direction, we assume $\mathcal{P} = \mathcal{NP}$ and show $\mathcal{NPC} \cap \mathcal{P} \neq \emptyset$. Since $\mathcal{NPC} \subseteq \mathcal{NP} = \mathcal{P}$, we have $\mathcal{NPC} = \mathcal{NPC} \cap \mathcal{P}$. Since $\mathcal{NPC} \neq \emptyset$ (see Theorem 57 below), we also have $\mathcal{NPC} \cap \mathcal{P} \neq \emptyset$.

To show the $\Leftarrow$ direction, we assume $\mathcal{NPC} \cap \mathcal{P} \neq \emptyset$ and show $\mathcal{P} = \mathcal{NP}$. Since $\mathcal{NPC} \cap \mathcal{P} \neq \emptyset$, there exists a problem P' such that $P' \in \mathcal{NPC}$ and also $P' \in \mathcal{P}$. To show $\mathcal{P} = \mathcal{NP}$, it suffices to show $\mathcal{NP} \subseteq \mathcal{P}$. We thus take an arbitrary problem $P \in \mathcal{NP}$ and show that $P \in \mathcal{P}$.

Since $P' \in \mathcal{NPC}$ and $P \in \mathcal{NP}$, by the definition of $\mathcal{NPC}$, $P \leq_P P'$. Since $P \leq_P P'$ and $P' \in \mathcal{P}$, by Theorem 54 we know $P \in \mathcal{P}$. □

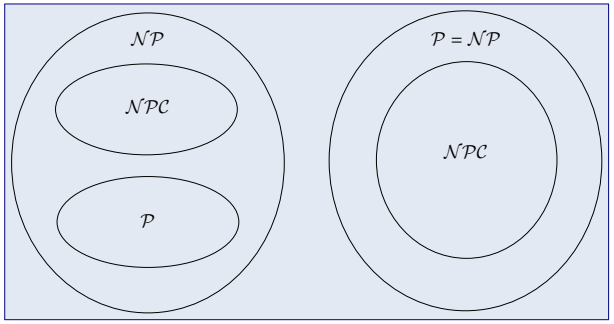


Figure 7.5.: $\mathcal{P}$ versus $\mathcal{NP}$ versus $\mathcal{NPC}$

A proof of $\mathcal{P} \neq \mathcal{NP}$ thus may focus on proving that $\mathcal{NPC} \cap \mathcal{P} = \emptyset$, i.e., that no $\mathcal{NP}$-complete problem can be deterministically decided in polynomial time.

Furthermore, we have the following result:

Theorem 56 ($\mathcal{NPC}$ versus $\mathcal{P}$) If $\mathcal{NPC} \cap \mathcal{P} \neq \emptyset$, then $\mathcal{NPC} \subseteq \mathcal{P}$.

In other words, if some $\mathcal{NP}$-complete problem can be deterministically decided in polynomial time, then all such problems can be deterministically decided in polynomial time.

Proof If $\mathcal{NPC} \cap \mathcal{P} \neq \emptyset$, then there exists some problem P' such that $P' \in \mathcal{NPC}$ and $P' \in \mathcal{P}$. To show $\mathcal{NPC} \subseteq \mathcal{P}$, we take an arbitrary problem $P \in \mathcal{NPC}$ and show $P \in \mathcal{P}$. Since $P \in \mathcal{NPC}$ and $\mathcal{NPC} \subseteq \mathcal{NP}$, we have $P \leq_P P'$. Since $P' \in \mathcal{P}$, by Theorem 54, also $P \in \mathcal{P}$. □

According to Theorems 55 and 56, we have two possibilities (see Figure 7.5):

1. either $\mathcal{NPC}$ and $\mathcal{P}$ are disjoint subsets of $\mathcal{NP}$; then $\mathcal{P} \neq \mathcal{NP}$ (and, as can be also shown, $\mathcal{NPC} \neq \mathcal{NP}$);
2. or $\mathcal{NPC}$ and $\mathcal{P}$ are not disjoint; then $\mathcal{NPC} \subseteq \mathcal{P} = \mathcal{NP}$ (there exists a different notion of reducibility than the one we have introduced in Definition 53, for which then even $\mathcal{NPC} = \mathcal{P} = \mathcal{NP}$ holds).

The quest to answer $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ can be thus reduced to the quest whether there exists some $P \in \mathcal{NPC}$ such that also $P \in \mathcal{P}$, i.e., whether there exists some $\mathcal{NP}$-complete problem that

can be decided in polynomial time. Of course, the answer would be immediately "no" if there did not exist any $\mathcal{NP}$-complete problems at all. However, as shown below, this is not the case.

aussagenlogische Formel

Definition 55 (Satisfiability) Let a *propositional formula* F be an expression formed according to the following grammar:

$$F ::= x_i \mid \neg F \mid F \wedge F \mid F \vee F$$

i.e., it is formed from boolean variables $x_0, x_1, \ldots$ by the application of the logical connectives *not*, *and*, and *or*.

erfüllbar

Erfüllbarkeitsproblem

A propositional formula F is *satisfiable* if there exists an assignment of the variables x_i in F to truth values which makes F true. The *satisfiability problem* is the problem to decide whether a propositional formula F is satisfiable.

Example 36 The propositional formula

$$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_1)$$

is satisfiable: we can construct the assignment $x_1 := \mathrm{T}, x_2 := \mathrm{F}$ and evaluate

$$(\mathrm{T} \vee \mathrm{F}) \wedge (\neg \mathrm{F} \vee \mathrm{T})$$

to T (true). On the other hand, the propositional formula

$$x_1 \wedge \neg x_2 \wedge (\neg x_1 \vee x_2)$$

is not satisfiable. □

The relevance of the satisfiability problem is illustrated by the following theorem which was proved by Stephen Cook in 1971:

Satz von Cook

Theorem 57 (*Cook's Theorem*) The satisfiability problem is $\mathcal{NP}$-complete, i.e., it is an element of problem class $\mathcal{NPC}$.

The satisfiability problem was the first problem that was be proved to be $\mathcal{NP}$-complete (we omit the proof); in the meantime also many other problems were shown to be of this kind (typically, by proving that the satisfiability problem is polynomial time-reducible to these problems), for instance:

Hamiltonian Path The (already discussed) problem to determine whether there exists a cyclic path in a graph that visits every node exactly once.

Traveling Salesman (Decision Version) The problem to decide whether there exists in a weighted graph a cyclic path of length less than a given maximum.

Graph Coloring The problem to decide, for a given graph and number of colors, whether the nodes of the graph can be colored such that no adjacent nodes get the same color.

Knapsack The problem to decide, given a collection of items with a certain "weight" and "value", which set of items is the most valuable one among those that can be packed into a "knapsack" with a certain weight limit.

Integer Programming Given an integer matrix A and vectors b, c, to determine, among all vectors x with $A \cdot x \leq b$, the one that maximizes the value $c \cdot x$.

Currently more than 3000 problems from widely different mathematical areas are known to be $\mathcal{NP}$-complete. Since each of these problems is polynomial time-reducible to any other of these problems, an algorithm for solving one problem can be applied (after appropriate problem reduction) to solve all other problems. This has immediate practical consequences: even if the satisfiability problem is $\mathcal{NP}$-complete, various *SAT-solvers* have been developed that solve this problem by heuristic methods efficiently for large classes of formulas; these solvers can then be applied to solve problems in many different areas, e.g., integrated circuit design, hardware and software verification, theorem proving, planning, scheduling, and optimization.

7.4. Complements of Problems

We will finally illuminate the quest for $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ from another angle, that of *co-problems* (i.e, complements of problems). We start with some simple results.

Theorem 58 (co-Problem Reducability) For all decision problems P and P' with $P \leq_{\mathrm{P}} P'$, we have

$$\overline{P} \leq_{\mathrm{P}} \overline{P'}$$

i.e., if P is polynomial time-reducible to P', than also the co-problem $\overline{P}$ is polynomial time-reducible to the co-problem $\overline{P'}$,

Proof We assume $P \leq_P P'$ and show $\overline{P} \leq_P \overline{P'}$. Thus we have to find a function f that can be computed by a deterministic Turing machine in polynomial time such that $\overline{P}(w) \Leftrightarrow \overline{P'}(f(w))$ which is equivalent to $P(w) \Leftrightarrow P'(f(w))$. From $P \leq_P P'$, we have exactly such a function f and are therefore done. □

Theorem 59 ($\mathcal{P}$ and Co-Problems) A problem P is in $\mathcal{P}$ if and only if its co-problem $\overline{P}$ is in $\mathcal{P}$:

$$\mathcal{P} = \{P \mid \overline{P} \in \mathcal{P}\}$$

In other words, it can be decided by a deterministic Turing machine in polynomial time whether some input has property P if and only if it can be decided by a deterministic Turing machine in polynomial time whether some input does *not* have property P.

Proof If a deterministic Turing machine can decide in polynomial time P, by reverting its answer it can decide in polynomial time $\overline{P}$, and vice versa. □

We now come to the core definition of this section.

Definition 56 (co-$\mathcal{NP}$) co-$\mathcal{NP}$ is the class of all problems whose complements are in $\mathcal{NP}$:

$$\text{co-}\mathcal{NP} := \{P \mid \overline{P} \in \mathcal{NP}\}$$

In other words, for every $P \in$ co-$\mathcal{NP}$ a nondeterministic Turing machine can decide in polynomial time whether some input is *not* in P.

In order to understand why we introduce the class co-$\mathcal{NP}$, it is important to note that, it is not self-evident, that if a *nondeterministic* Turing machine M decides a problem P in polynomial time, that also the complement $\overline{P}$ can be decided in polynomial time. The reason is that M accepts some input w if there is at least one accepting run with answer "yes", i.e., there may be

also runs of M with answer "no" that do not accept w. If we construct a Turing machine $\overline{M}$ that just reverts the "yes" and "no" answers from M, then also $\overline{M}$ may accept w, but then the language of $\overline{M}$ is not the complement $\overline{P}$. In order to really decide $\overline{P}$, the Turing machine $\overline{M}$ may accept an input w only, if M does for input w not have *any* run with a "yes" answer; since the number of runs of M may be exponential in the length of w, it is not clear how this can be decided by any nondeterministic Turing machine in polynomial time.

However, we have the following result:

Theorem 60 ($\mathcal{P}$ versus $\mathcal{NP}$ and co-$\mathcal{NP}$)

$$\mathcal{P} \subseteq \mathcal{NP} \cap \text{co-}\mathcal{NP}$$

In other words, for every problem in $\mathcal{P}$, both the problem and its complement can be decided by a nondeterministic Turing machine in polynomial time.

Proof We take arbitrary $P \in \mathcal{P}$ and show $P \in \mathcal{NP} \cap$ co-$\mathcal{NP}$. Since $P \in \mathcal{P}$, also $P \in \mathcal{NP}$. It thus suffices to show $P \in$ co-$\mathcal{NP}$. By Theorem 59, we know $\overline{P} \in \mathcal{P}$. Since $\mathcal{P} \subseteq \mathcal{NP}$, we know $\overline{P} \in \mathcal{NP}$ and thus $P \in$ co-$\mathcal{NP}$. □

The relevance of co-$\mathcal{NP}$ to the question $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ is illustrated by the following theorem:

Theorem 61 ($\mathcal{NP}$ versus co-$\mathcal{NP}$) If $\mathcal{NP} \neq$ co-$\mathcal{NP}$, then $\mathcal{P} \neq \mathcal{NP}$.

Proof We assume $\mathcal{NP} \neq$ co-$\mathcal{NP}$ and show $\mathcal{P} \neq \mathcal{NP}$. We define the property $C(S)$ of a class of problems S as

$$C(S) :\Leftrightarrow S = \{P \mid \overline{P} \in S\}$$

From $\mathcal{NP} \neq$ co-$\mathcal{NP}$, we know $\neg C(\mathcal{NP})$. From Theorem 59, we know $C(\mathcal{P})$. Thus $\mathcal{P} \neq \mathcal{NP}$.□

To show $\mathcal{P} \neq \mathcal{NP}$, it thus suffices to show $\mathcal{NP} \neq$ co-$\mathcal{NP}$.

By Theorems 60 and 61, we thus have the following two possibilities for the relationships between the classes (see Figure 7.6):

1. either $\mathcal{NP} \neq$ co-$\mathcal{NP}$ and $\mathcal{P} \subseteq \mathcal{NP} \cap$ co-$\mathcal{NP}$ and $\mathcal{P} \neq \mathcal{NP}$,

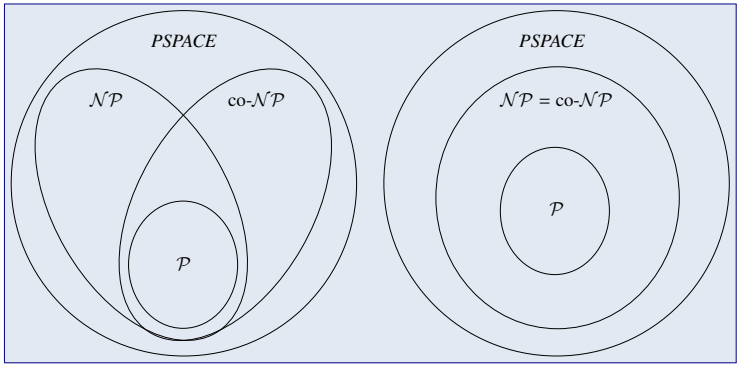


Figure 7.6.: $\mathcal{NP}$ versus co-$\mathcal{NP}$

2. or $\mathcal{NP}$ = co-$\mathcal{NP}$ and $\mathcal{P} \subseteq \mathcal{NP}$ = co-$\mathcal{NP}$ ($\mathcal{P} = \mathcal{NP}$ may or may not hold).

Since also co-$\mathcal{NP} \subseteq$ *PSPACE* can be shown, all classes are contained in *PSPACE*.

Furthermore, we can relate the question $\mathcal{NP} \stackrel{?}{=}$ co-$\mathcal{NP}$ to the class $\mathcal{NPC}$.

Theorem 62 (co-$\mathcal{NP}$ versus $\mathcal{NPC}$)

$$\mathcal{NP} \neq \text{co-}\mathcal{NP} \Leftrightarrow \forall P \in \mathcal{NPC} : \overline{P} \notin \mathcal{NP}$$

In other words, $\mathcal{NP}$ differs from co-$\mathcal{NP}$, if and only if the complements of all $\mathcal{NP}$-complete problems cannot be decided by a nondeterministic Turing machine in polynomial time.

Proof We show both directions of the proof.

$\Leftarrow$ We assume $\mathcal{NP}$ = co-$\mathcal{NP}$ and show that there exists some $P \in \mathcal{NPC}$ with $\overline{P} \in \mathcal{NP}$. Take arbitrary $P \in \mathcal{NPC}$. Since $\mathcal{NPC} \subseteq \mathcal{NP}$ and $\mathcal{NP}$ = co-$\mathcal{NP}$, also $P \in$ co-$\mathcal{NP}$ and thus $\overline{P} \in \mathcal{NP}$.

$\Rightarrow$ We assume, for some $P \in \mathcal{NPC}$, $\overline{P} \in \mathcal{NP}$, and show $\mathcal{NP}$ = co-$\mathcal{NP}$.

To show $\mathcal{NP} \subseteq$ co-$\mathcal{NP}$, we take arbitrary $Q \in \mathcal{NP}$ and show $Q \in$ co-$\mathcal{NP}$. Since $P \in \mathcal{NPC}$ and $Q \in \mathcal{NP}$, we have $Q \leq_\text{P} P$ and thus by Theorem 58 also $\overline{Q} \leq_\text{P} \overline{P}$. Since $\overline{P} \in \mathcal{NP}$, by Theorem 54 we also have $\overline{Q} \in \mathcal{NP}$ and thus $Q \in$ co-$\mathcal{NP}$.

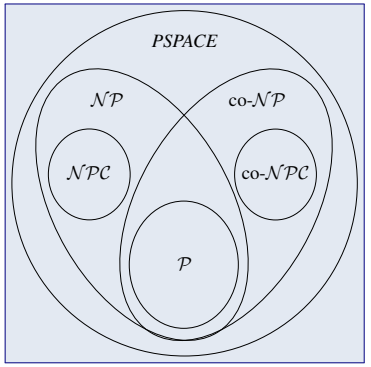


Figure 7.7.: $\mathcal{NP}$ versus co-$\mathcal{NP}$ (Conjecture)

To show co-$\mathcal{NP} \subseteq \mathcal{NP}$, we take arbitrary $Q \in$ co-$\mathcal{NP}$ and show $Q \in \mathcal{NP}$. Since $Q \in$ co-$\mathcal{NP}$, we have $\overline{Q} \in \mathcal{NP}$. Since $P \in \mathcal{NPC}$ and $\overline{Q} \in \mathcal{NP}$, we have $\overline{Q} \leq_\text{P} P$ and thus by Theorem 58 also $Q \leq_\text{P} \overline{P}$. Since $\overline{P} \in \mathcal{NP}$, by Theorem 54 we also have $Q \in \mathcal{NP}$. □

By Theorem 62, we can refine our knowledge about the first possibility shown in Figure 7.6 about the relationships between the various complexity classes (see Figure 7.7 where co-$\mathcal{NPC}$ denotes the class of the complements of all problems in $\mathcal{NPC}$): $\mathcal{NP} \neq$ co-$\mathcal{NP}$ holds, if and only if $\mathcal{P} \subseteq \mathcal{NP} \cap$co-$\mathcal{NP}$ (thus $\mathcal{P} \neq \mathcal{NP}$) and $\mathcal{NPC} \subseteq \mathcal{NP} \backslash$co-$\mathcal{NP}$ and co-$\mathcal{NPC} \subseteq$ co-$\mathcal{NP} \backslash \mathcal{NP}$. This is the most likely situation as conjectured by most researchers.

On the other hand, if there existed some problem $P \in \mathcal{NPC}$ such that also $\overline{P} \in \mathcal{NP}$, i.e., an $\mathcal{NP}$-complete problem whose complement is decidable by a nondeterministic Turing machine in polynomial time, we would have $\mathcal{NPC}$ = co-$\mathcal{NPC}$ and consequently $\mathcal{NP}$ = co-$\mathcal{NP}$ such that $\mathcal{P} = \mathcal{NP}$ might hold. However, no such problem could be found yet, which may be considered as evidence for $\mathcal{P} \neq \mathcal{NP}$ (actually, for no $P \in \mathcal{NPC}$ it is even known whether $\overline{P} \in \mathcal{NP}$ or not).

Conversely, if for a problem P both $P \in \mathcal{NP}$ and $\overline{P} \in \mathcal{NP}$ can be shown, this may be considered as evidence for $P \notin \mathcal{NPC}$, i.e., that the problem is *not* $\mathcal{NP}$-complete (because otherwise we would have $\mathcal{NPC}$ = co-$\mathcal{NPC}$ and thus $\mathcal{P} = \mathcal{NP}$ might hold). This is e.g. the case for the problem of *integer factorization*, i.e., the problem of deciding whether a given integer can be decomposed into non-trivial factors. This question is also of practical relevance, since the security of *public-key cryptography* depends on the assumption that integer factorization is a "hard" problem. However, since this problem might not be $\mathcal{NP}$-complete, it might be not be

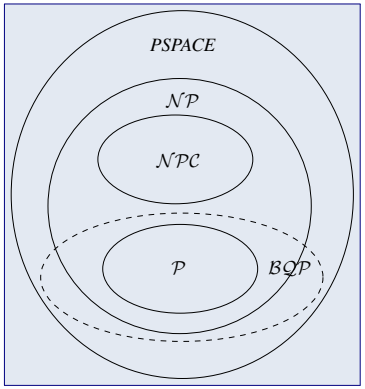


Figure 7.8.: The Quantum Computing Class $\mathcal{BQP}$ (Conjecture)

hard enough, i.e., it might become eventually feasible to factorize large integers (the currently best known deterministic algorithms fall into a complexity class between $\mathcal{P}$ and *EXPTIME*; it is conjectured that the integer factorization problem is not in $\mathcal{P}$).

These questions are also relevant to the emerging field of *quantum computing*. In 1994 Peter Shor discovered an algorithm for quantum computers that solves the integer factorization problem in polynomial time, thus quantum computers might in some future break public-key cryptography. Indeed quantum algorithms have been devised that solve various problems asymptotically faster than the fastest known classical algorithms. For the class $\mathcal{BQP}$ of *bounded error quantum polynomial time* (the class of problems solvable by a quantum computer in polynomial time with limited error probability), it is currently known that

$$\mathcal{P} \subseteq \mathcal{BQP} \subseteq \mathit{PSPACE}$$

However, up to now no quantum algorithm is known that solves any $\mathcal{NP}$-complete problem in polynomial time. In fact, most researchers believe that the class $\mathcal{NPC}$ and the class $\mathcal{BQP}$ are disjoint and that $\mathcal{BQP}$ and $\mathcal{NP}$ are incomparable. This conjecture is depicted in Figure 7.8.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 3rd edition, 2009.

[3] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison Wesley, 2nd edition, 1994.

[4] Dirk W. Hoffmann. *Theoretische Informatik*. Carl Hanser, Munich, Germany, 2009.

[5] Johan E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Automata Theory, Languages, and Computation*. Pearson Education, Boston, MA, USA, 3rd edition, 2007.

[6] Manuel Kauers and Peter Paule. *The Concrete Tetrahedron — Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates*. SpringerWienNewYork, 2011.

[7] Donald E. Knuth. *The Art of Computer Programming: Volume 1 (Fundamental Algorithms)*. Addison Wesley, 2nd edition, 1973.

[8] Peter Rechenberg and Gustav Pomberger, editors. *Informatik-Handbuch*. Carl Hanser Verlag, 4th edition, 2006.

[9] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Cengage Learning, Boston, MA, USA, 2nd edition, 2006.

[10] Franz Winkler. Formale Grundlagen der Informatik 2 (Theoretische Informatik). Skriptum WS 2005/2006, Research Institute for Symbolic Compoutation (RISC), Johannes Kepler University, Linz, Austria, 2005.

Index

Q

R

S

T

U

V

W