Fachhochschule Südwestfalen
University of Applied Sciences
South Westphalia University of Applied Sciences
Campus Hagen, Germany
Dep. Business Engineering

Algorithmics and Optimization

Lecture Notes in Business Informatics (2nd Semester)

Andreas de Vries

Version: September 15, 2015

Contents

I Foundations of algorithmics **8**

1 Elements and control structures of algorithms **9**
1.1 Mathematical notation 9
1.2 The basic example: Euclid's algorithm 10
1.3 The elements of an algorithm 11
1.4 Control structures 12
1.5 Definition of an algorithm 15

2 Algorithmic analysis **16**
2.1 Correctness ("effectiveness") 16
2.2 Complexity to measure efficiency 17
2.3 Summary 23

3 Recursions **24**
3.1 Introduction 24
3.2 Recursive algorithms 25
3.3 Searching the maximum in an array 26
3.4 Recursion versus iteration 27
3.5 Complexity of recursive algorithms 28
3.6 The towers of Hanoi 31

4 Sorting **34**
4.1 Simple sorting algorithms 34
4.2 Theoretical minimum complexity of a sorting algorithm 35
4.3 A recursive construction strategy: Divide and conquer 36
4.4 Fast sorting algorithms 37
4.5 Comparison of sort algorithms 43

5 Searching with a hash table **44**
5.1 Hash values 44
5.2 The hashing principle 48
5.3 Collisions 49

II Optimization and Networks **54**

6 Optimization problems **55**
6.1 Examples 55
6.2 The general structure of optimization problems 56
6.3 Approaches to solve optimization problems 60

7 Graphs and shortest paths **63**
7.1 Basic definitions . . . 63
7.2 Representation of graphs . . . 64
7.3 Traversing graphs . . . 65
7.4 Cycles . . . 67
7.5 Shortest paths . . . 69

8 Dynamic Programming **75**
8.1 An optimum-path problem . . . 75
8.2 The Bellman functional equation . . . 79
8.3 Production smoothing . . . 79
8.4 The travelling salesman problem . . . 84

9 Simplex algorithm **87**
9.1 Mathematical formulation . . . 87
9.2 The simplex algorithm in detail . . . 88
9.3 What did we do? or: Why simplex? . . . 91
9.4 Duality . . . 92

10 Genetic algorithms **96**
10.1 Evolutionary algorithms . . . 96
10.2 Basic notions . . . 96
10.3 The “canonical” genetic algorithm . . . 98
10.4 The 0-1 knapsack problem . . . 100
10.5 Difficulties of genetic algorithms . . . 101
10.6 The traveling salesman problem . . . 103
10.7 Axelrod’s genetic algorithm for the prisoner’s dilemma . . . 104
10.8 Conclusions . . . 106

Appendix **107**

A Mathematics **108**
A.1 Exponential and logarithm functions . . . 108
A.2 Number theory . . . 108
A.3 Searching in unsorted data structures . . . 110

B Dictionary for mathematics and computer science **114**

C Arithmetical operations **117**

Bibliography **120**

Dear reader,

perhaps you are surprised finding these lecture notes written in English (or better: in a scientific idiom which is very similar to English). We have decided to write them in English for the following reasons. Firstly, any computer scientist or information system technologist has to read a lot of English documents, web sites, text books — particularly if he or she wants to get to know innovative issues. So this is the main reason: training for non-natively English speaking students. Secondly, foreign students and international institutions shall benefit from these notes. Thirdly, the notes offer a convenient way to learn the English terminology corresponding to the German one given in the lecture.

To help you to learn the English terminology, you will find a small dictionary at the end of these notes, presenting some of the expressions most widely used in computer sciences and mathematics, along with their German translations. In addition, there are listed some arithmetical terms in English and in German.

You might as well get surprised to find another human language — *mathematics*. Why mathematics in a book about algorithmics? Algorithms are, in essence, *applied* mathematics. Even if they deal with apparently "unmathematical" subjects such as manipulating strings or searching objects, mathematics is the basis. To mention just a few examples: the classical algorithmic concept of recursion is very closely related to the principle of mathematical induction; rigorous proofs are needed for establishing the correctness of given algorithms; running times have to be computed.

The contents of these lecture notes spread a wide range. On the one hand they try to give the basic knowledge about *algorithmics*, such that you will learn the following questions: What is an algorithm and what are its building blocks? How can an algorithm be analyzed? How do standard well-known algorithms work? On the other hand, these lecture notes introduce into the wide and important field of *optimization*. Optimization is a basic principle of human activity and thinking, it is involved in the sciences and in practice. It mainly deals with the question: How can a solution to a given problem under certain constraints be achieved with a minimum cost, be it time, money, or machine capacity? Optimization is a highly economical principle. However, any solution of an optimization problem is a list of instructions, such as "do this, then do that, but only under the condition that ...," i.e., an algorithm — the circle is closed.

So we think optimization to be one of the basic subjects for you as a student of business information systems, for it will be one of your main business activities in the future. Surely, no lecture can give an answer to all problems which you will be challenged, but we think that it is important to understand that *any* optimization problem has a basic structure — it is the structure of a given optimization problem that you should understand, because then you may solve it in a more efficient way (you see, another optimization problem).

Of course, a single semester is much too short to mention all relevant aspects. But our hope is that you gain an intuitive feeling for the actual problems and obstacles. For this is what you really must have to face future challenges — *understanding*.

Hagen September 15, 2015

Andreas de Vries

Introduction

The central notion of computer science is "algorithm," not "information." An algorithm is a detailed and finite recipe to solve a problem. Algorithms always act on data. So algorithms and data belong together. This simple fact is most consequently realized in object-oriented approach: Here algorithms are realized in so-called "methods" and data are named "attributes." They both form a unity called "object."

The right choice of algorithms and data structures therefore is the most important step to solve a problem with the help of a computer. The subject of this script is the systematic study of algorithms and data in different kinds of application.

Mathematics, algorithms, and programming

The following diagram is due to Güting [19]. It describes analog notions in different scientific areas. The roles that functions and algebras play in mathematics corresponds to the algorithms and the data structures in the area of algorithmics, rsp., and to the notions of method and attributes or objects as a whole in (oo-) programming.

Area	Operation	Operand structure
mathematics	function	algebra
	↓	↓
algorithmics	algorithm	data structure
	↓	↓
programming	procedure, function	data type
	method	attributes
	object	

An *algebra* determines a set of objects and its arithmetic, i.e. the way in which the objects can be "calculated." It defines an operations called "addition," one called "scalar multiplication," and one called "multiplication." An example for an algebra is a vector space, where the objects are vectors, where multiplication is the vector product, and addition and scalar multiplication are as usual.

Mathematics		
Algebra	**Objects**	**Arithmetic operations**
number algebra	numbers x, y	$x+y, x-y, x\cdot y, x/y$
vector algebra	vectors v, w	$v \pm w, v \cdot w, x \cdot v$ (for $x \in \mathbb{R}$)
matrix algebra	matrices A, B	$A \pm B, A \cdot B, x \cdot Ax \cdot v$ (for $x \in \mathbb{R}$)

Algorithms and programming languages

Computer technology has undergone and still undergoes a very rapid development. According to the empirical "Moore's law" in its generalized version the computing power per unit cost

doubles roughly every 24 months. So the applicability and the abilities of computers are growing more and more. They fly aeroplanes and starships, control power stations and cars, find and store information, or serve as worldwide communication devices. Over the last three decades, computers have caused a technological, economic, and social revolution which could be hardly foreseen.

Parallel to the technology changes, and in part having enabled them, there is a development of various programming languages. From the first "higher" programming languages of the 1950's for scientific and business-oriented computations, like Fortran and COBOL, to internet-based languages like Java or PHP, every new field of activity made available some new programming languages specialized in it.

In view of this impressive and enormous developments, the question may be raised: Is there anything that remains constant during all these changes? Of course, there are such constants, and they were to a great part stated already *before* the invention of the first computers in the 1930's, mainly achieved by the mathematicians Gödel, Turing, Church, Post and Kleene: These are the fundamental laws underlying any computation and hence any programming language. These fundamental laws of *algorithms* are the subject of this book, not a particular programming language.

However, in this book the study of algorithms is done on the background and influenced by the structure of Java, one of the most elaborated and widely used programming languages. In particular, the pseudocode to represent algorithms is strongly influenced by the syntax of Java, although it should be understandable without knowing Java.

References

The literature on algorithmics and optimization is immense. The following list only is a tiny and uncomplete selection.

- T.H. Cormen et al.: *Introduction to Algorithms* [5] – classical standard reference, with considerable breadth and width of subjects. A *must* for a computer scientist.

- R. L. Graham, D. E. Knuth & O. Patashnik: *Concrete Mathematics* [16] – very good reference for the mathematical foundations of computer programming and algorithmics. ("concrete" is a blending of "*con*tinuous" and "dis*crete*"); One of the authors, Knuth, is the inventor of the fabulous text-writing system TEX, the essential basis of LATEX these lecture notes are set with...

- H. P. Gumm & M. Sommer: *Einführung in die Informatik.* [18] – broad introduction to computer science, with emphasis on programming.

- D. Harel & Y. Feldman: *Algorithmik. Die Kunst des Rechnens* [21] – gives a good overview over the wide range of algorithms and the underlying paradigms, even mentioning quantum computation.

- D. W. Hoffmann: *Theoretische Informatik* [24] – broad introduction to theoretical computer science.

- F. Kaderali & W. Poguntke: *Graphen, Algorithmen, Netze* [26]. Basic introduction into the theory of graphs and graph algorithms.

- T. Ottmann & P. Widmayer: *Algorithmen und Datenstrukturen* [33] – classical standard reference.

- A. Barth: *Algorithmik für Einsteiger* [1] – a nice book explaining principles of algorithmics.
- W. Press et al.: *Numerical Recipes in C++* [36] – for specialists, or special problems. To lots of standard, but also rather difficult problems, there is given a short theoretical introduction and descriptions of efficient solutions. Requires some background in mathematics.

For further reading in German I recommend [19, 23, 38, 44].

Part I

Foundations of algorithmics

Chapter 1

Elements and control structures of algorithms

1.1 Mathematical notation

Definition 1.1. For any real number x we denote by $\lfloor x \rfloor$ the greatest integer which is less than or equal to x, or more formally:

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} |\ n \leqq x\}. \tag{1.1}$$

The $\lfloor \ldots \rfloor$-signs are called *floor-brackets* or *lower Gauß-brackets*. □

For example we have $\lfloor 5.43 \rfloor = 5$, $\lfloor \pi \rfloor = 3$, $\lfloor \sqrt{2} \rfloor = 1$, $\lfloor -5.43 \rfloor = -6$. Note that for two positive integers $m, n \in \mathbb{N}$ we have

$$\left\lfloor \frac{m}{n} \right\rfloor = m \text{ div } n,$$

where "div" denotes integer division. In Java, we have for two integer variables `int m,n`

$$\left\lfloor \frac{m}{n} \right\rfloor = \begin{cases} m/n & \text{if } m \cdot n \geqq 0, \\ m/n - 1 & \text{if } m \cdot n < 0. \end{cases}$$

1.1.1 The modulo operation and %

In the mathematical literature you find the notation

$$k = n \bmod m, \qquad \text{or} \quad k \equiv n \bmod m.$$

For positive numbers m and n, this means the same as "%". However, for $n < 0$ and $m > 0$, there is a difference:

$$n \bmod m = (m + n \ \%\ m) \ \%\ m \qquad \text{if } n < 0 \text{ and } m > 0. \tag{1.2}$$

This difference stems from the fact that "modulo m" mathematically means a consequent arithmetic only with numbers k satisfying $0 \leqq k < m$, whereas "%" denotes the remainder of an integer division.

n	$\cdots$	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
$n \bmod 3$	$\cdots$	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	$\cdots$
$n \ \%\ 3$	$\cdots$	-2	-1	0	-2	-1	0	1	2	0	1	2	0	1	2	0	1	$\cdots$

For instance, $-5 \ \%\ 3 = -(5 \ \%\ 3) = -2$, but $-5 \bmod 3 = 1$. Thus the result of the *modulo* operation always gives a nonnegative integer $k < 3$, cf. [16, §3.4].

1.2 The basic example: Euclid's algorithm

The notion of algorithm is basic to all of computer programming. The word "algorithm" itself is quite interesting. It comes from the name of the great Persian mathematician, Abu Abdullah abu Jafar Muhammad ibn Musa *al-Khwarizmi* (about 780 – about 850) — literally "Father of Abdullah, Jafar Mohammed, son of Moses, native of Khwarizm." The Aral sea in Central Asia was once known as the Lake Khwarizm, and the Khwarizm region is located south of that sea. Al-Khwarizmi wrote the celebrated book *Kitab al-jabr wa'l-muqabala* ("Rules of restoring and equating"), which was a systematic study of the solutions of linear and quadratic equations. From the title of this book stems the word "algebra" *(al-jabr)*. For further details see [27, 28].

Very famous, and older than 2300 years, is *Euclid's algorithm*, called after the Greek mathematician Euclid (350–300 b.c.). Perhaps he did not invent it, but he is the first one known to have written it down. The algorithm is a process for finding the greatest common divisor of two numbers.

Algorithm 1.2. (Euclid's algorithm) Given two positive integers m and n, find their *greatest common divisor* gcd, that is, the largest positive integer that evenly divides both m and n.

E1. [Exchange m and n] Exchange $m \leftrightarrow n$.

E2. [Reduce n modulo m] Assign to n its value modulo m:

$$n \leftarrow n \,\%\, m.$$

(Remember: 'modulo' means 'remainder of division'; *after* the assignment we have $0 \leqq n < m$.)

E3. [Is n greater than zero?] If $n > 0$ loop back to step E1; if $n \leqq 0$, the algorithm terminates, m is the answer.

Let us illustrate by an example to see how Euclid's algorithm works. Consider $m = 6$, $n = 4$.

- Step E1 exchanges m and n such that $m = 4$ and $n = 6$; step E2 yields the values $m = 4$, $n = 2$; because in E3 still $n > 0$, step E1 is done again.
- Again arriving in E1, m and n are exchanged, yielding the new values $m = 2$, $n = 4$; E2 yields the new value $n = 0$, and still $m = 2$; E3 tells us that $m = 2$ is the answer.

Thus the greatest common divisor of 6 and 4 is 2,

$$\gcd(6,4) = 2.$$

A first observation is that the verbal description is not a very convenient technique to describe the effect of an algorithm. Instead, we will create a value table denoting the values depending on the time, so to say the "evolution of values" during the algorithm time flow of the algorithm, see 1.1.

1.2.1 Pseudocode

A convenient way to express an algorithm is *pseudocode*. This is an artificial and informal language which is similar to everyday English, but also resembles to higher-level programming languages such as Java, C, or Pascal. (In fact, one purpose of pseudocode is just to enable the direct transformation into a programming language; pseudocode is the "mother" of all programming languages). Euclid's algorithm in pseudocode reads as follows:

Step	**Value after step**		
	$n > 0$?	m	n
Start		6	4
E1		4	6
E2			2
E3	yes		
E1		2	4
E2			0
E3	no	$\boxed{2}$	

Table 1.1: Table of values during the algorithm flow

```
euclid (m,n) {
  while ( n > 0 ) {
      m ↔ n;
      n ← n % m;
  }
  return m;
}
```

By convention, any assignmet is terminated by a semicolon (;). This is in accordance with most of the common programming languages (especially Java, C, C++, Pascal, PHP). Remarkably, Euclid's algorithm is rather short in pseudocode. Obviously, pseudocode is a very effective way to represent algorithms, and we will use it throughout this script.

We use the following conventions in our pseudocode.

1. In the first line the name of the algorithm appears, followed by the required parameters in parentheses: euclid (m,n)
2. Indention indicates block structure. For example, the body of the *while*-loop only consists of one instruction. Often we will indicate block structure in addition by {...} (as in Java or C), but it could easily be read as `begin` ... `end` (as in Pascal)).
3. We use as control structure key words only *while*, *for*, and *if* ... *else* as in the common programming languages (see below for details on control structures)
4. Comments are indicated by the double slash //. It means that the rest of the line is a comment.
5. We will use the semicolon (;) to indicate the end of an instruction.

With the Euclidean algorithm we will explore what the basic element are out of which a general algorithm can be built: the possible *operations*, the *assigment*, and three *control structures*. With these elements we are able to define what actually an algorithm *is*.

1.3 The elements of an algorithm

1.3.1 Operations

The possible *operations* are general mathematical functions

$$f : D \to R \tag{1.3}$$

with the "domain of definition" $D \subset \mathbb{R}^d$ and the "range" $R \subset \mathbb{R}^r$ and $d, r \in \mathbb{N} \cup \{\pm\infty\}$. For instance, the modulo operation is given by the function

$$f : \mathbb{N}^2 \to \mathbb{N}, \quad f(m,n) = m \,\%\, n.$$

Here $d = 2$ and $r = 1$.

Even non-numerical operations such as "assignment of a memory address" or "string addition" (concatenation) are possible operations, the sets D and R only have to be defined appropriately. (In the end: All strings are natural numbers, [3] p.213.) Also Boolean functions evaluating logical expressions (such as $x < y$) are possible.

1.3.2 Instructions

An *instruction* is an elementary command to do an "action". We will be dealing only with three instructions: *input*, *output*, and the assignment.

The instructions *input* and *output* denote the methods which manage the flow of data into and out of the system. They turn out to be the entrance and the exit door of an algorithm. Both are *methods* which process "letters" of a given "alphabet." In Euclid's algorithm, there are two input "letters", namely m and n, and the "alphabet" is the set $\mathbb{N}$ of positive integers; the output is one "letter" n. ($D = \mathbb{N} \times \mathbb{N}$, $R = \mathbb{N}$).

Often the instruction *input* is replaced by the name of the input parameter list of the algorithm, e.g. "abc$(m,n;\ldots)$". If the algorithm has no input parameters, we can write it with empty parentheses such as "abs()." This notation makes sense especially if there is no further input data during the algorithm performance. Analogously, the key word *return* is frequently used instead of *output*.

The *assignment* is denoted by the arrow $\leftarrow$. The assignment $m \leftarrow n$ assigns the "value" of the right side n to the "variable" m on the left side. The right side may be an operation with a well-defined result, e.g., a subtraction:

$$m \leftarrow 4 - 2.$$

Here m is assigned the value 2. A necessary condition is of course that the value of the right side is well-defined. An assignment $m \leftarrow n$ in which n is a variable which has no definite value is not valid. If on the other hand m has the value 4, say, then the assignment

$$m \leftarrow m - 2$$

makes sense, meaning that m gets the value 2. Note that the order of assignment instruction is important: If, e.g., n has initial value 4 then the instruction sequence "$m \leftarrow n$; $n \leftarrow 1$;" is quite different from the sequence "$n \leftarrow 1$; $m \leftarrow n$;"

We will often deal with the *exchange* instruction "$\leftrightarrow$". It can be realized as a sequence of instruction with a new ("intermediate") variable t:

$$\text{"}m \leftrightarrow n;\text{"} \qquad \text{is defined as} \qquad \text{"}t \leftarrow m;\ m \leftarrow n;\ n \leftarrow t;\text{"} \tag{1.4}$$

Sometimes we will use multiple assignments $m \leftarrow n \leftarrow t$; it means that both variables m and n are assigned the value of t.

1.4 Control structures

Only one instruction can be executed at a time. It is the order of execution that can vary, determined by the so-called *control structure*. There are five types of flow control.

1.4.1 Sequence

It is the simplest of the control structures. Here each instruction is simply executes one after the other. In pseudocode, the sequence structure simply reads

instruction 1;
...;
instruction n;

1.4.2 Selection, choice

The selection is used to choose among alternative courses of instructions. In pseudocode it reads

if (condition) *{*
    instruction 1;
    ...;
    instruction m
} else {
    instruction 1;
    ...;
    instruction n;
}

Here a *condition* is a logical proposition being either `true` or `false`. It is also called a *Boolean expression*. If it is `true`, the instructions i_1, ..., i_m are executed, if it is `false`, the instructions e_1, ..., e_n are executed. If $n = 0$, the `else`-branch can be omitted completely. An example is given in Euclid's algorithm

if $(m < n)$ *{*
    $m \leftrightarrow n$
}

1.4.3 Loop, repetition

In a loop a given sequence of instruction is repeated as long as a specific condition is *true*. In pseudocode it is expressed by

while (condition) *{*
    instruction 1;
    ...;
    instruction n;
}

If the loop is performed a definite number of times, we also use the *for*-statement:

for $(i = 1$ *to* $m)$ *{*
    instruction 1;
    ...;
    instruction n;
}

It means that the instruction block is executed m times.

1.4.4 Subroutine calls

Subroutines are algorithms which can be invoked in another algorithm by simply calling its name and inputting the appropriate parameters, and whose results can be used. The pseudocode of a subroutine call may look like:

myAlgorithm(m,n) *{*
    $k = m *$ subroutine(n);
    return k;
}

A special subroutine call is the "recursion" which we will consider in more detail below. The terminology varies, subroutines are also be known as routines, procedures, functions (especially if they return results) or methods.

1.4.5 Exception handling, try-catch structure

Exception handling is a construct to handle the occurrence of some exception which prevents the algorithm to proceed in a well-defined way. Such an exception may be a division by zero which may occur during its flow of execution. The pseudocode of an exception handling is given as follows:

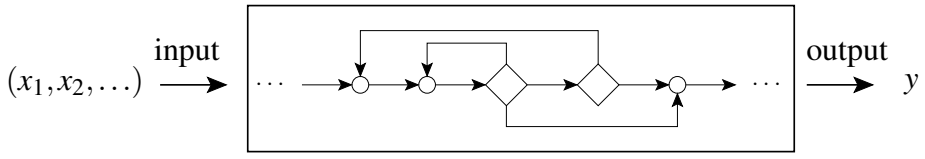
try {
    instruction 1;
    ...;
    instruction n;
} catch (exception1 A) *{*
    instruction A;
} catch (exception2 B) *{*
    instruction B;
}

Here the try-block contains the instructions of the algorithm. These instructions are monitored to perform correctly. If now an exception occurs during its execution, it is said to be "thrown," and according to its nature it is "catched" by one of the following catch-blocks, i.e., the execution flow is terminated and jumps to the appropriate catch-block. The sequence of catch-blocks has to be arranged from the special cases to more general cases. For instance, the first catched exception may be an arithmetic exception such as division by zero, the next one a more general runtime exception such as number parsing of a non-numeric input, or a IO exception such as trying to read a file which is not present, and so on to the catch-block for the most possible exception. In Java, the most general exception is an object of the class `Exception`.

1.5 Definition of an algorithm

Now we are ready to define a general algorithm. In essence it calculates to every input a deterministic output by executing finite instructions. More formally:

Definition 1.3. An *algorithm* is a finite sequence of instructions, constructed by one of the control structures, which takes a (possibly empty) set of values as *input* and produces a unique set of values as *output* in a finite time.



The output is a correct answer to a given "relevant" problem. □

The output usually should not be an empty set, for then the algorithm has no output and is needless.[1] It is interesting to note that *any* algorithm can be expressed by the first three control structures sequence, selection, and loop. This is a theoretical result from the 1960's.[2] This definition is equivalent to another theoretical concept, the *Turing machine*. In principle, this is a computing device executing a program. It is the theoretical model of a general computer program and was originally studied by Turing in the 1930's.[3]

So this is an algorithm. Further synonymic notions are *routine*, *process*, or *method*. By our definition, an algorithm thus has the following important properties:

1. *(finite)* An algorithm always terminates after a finite number of steps. Euclid's algorithm for instance *is* finite, because in step E1 m is always assigned the maximum value $\max(m,n)$ of m and n, whereas in step E2 m *decreases* properly. So for each initial pair (m,n) the algorithm will definitely terminate. (Note, however, that the number of steps can become arbitrarily large; certain huge choices for m and n could cause step E1 be executed more than a million times.) It can be proved[4] that Euclid's algorithm for two natural numbers m and n takes at most N loops where is the greatest natural number with
$$N \leqq 2.078 \ln[\max(m,n)] + 0.6723,$$
2. *(definite)* Each step of an algorithm is defined precisely. The actions to be carried out must be rigorously and unambiguously specified for each case.
3. *(elementary)* All operation must be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper. Operations may be clustered to more complex operations, but in the end they must be definitely reducible to elementary mathematical operations.
4. *(input)* An algorithm has zero or more inputs, i.e. data which are manipulated.
5. *(ouput or return)* An algorithm has one or more returns, i.e. information gained by the data and the algorithm.

---

[1] *Tue Gutes und rede darüber!*

[2] C. Bohm & G. Jacopini (1966): 'Flow diagrams, Turing machines, and languages with only two formation rules', *Comm. ACM* **9** (5), 336–371.

[3] Although there are generalizations of the "serial" algorithms defined here, e.g., parallel, distributed, and quantum algorithms, it is known today that only quantum algorithms might be in fact more powerful than Turing machines [7, §12].

[4] See [29, §4.5.3, Corollary L (p.360)]; more accurately $N \leqq \log_\phi[(3-\phi)\cdot\max(m,n)]$, where ϕ is the "golden ratio" $\phi = (1+\sqrt{5})/2$.

Chapter 2

Algorithmic analysis

There are two properties which have to be analysed when designing and checking an algorithm. On the one hand it has to be *correct*, i.e., it must answer the posed problem "effectively." Usually to demonstrate the correctness of an algorithm is a difficult task, it requires a mathematical proof. On the other hand, an algorithm should find a correct answer *efficiently*, i.e., as fast as possible with the minimum memory space.

2.1 Correctness ("effectiveness")

A major task to do for algorithms is to show that it is *correct*. It is not sufficient to test the algorithm with selected examples: If a test fails, the algorithm is indeed shown to be incorrect — but if some tests are o.k., the algorithm may be false nonetheless. A famous example is the function ("Euler equation")

$$f(n) = n^2 + n + 41. \tag{2.1}$$

If one asserts that $f(n)$ yields a prime number, one can test it for $n = 0, 1, 2, 3$, yes even for $n = 10$ $f(n) = 151$ (this is a prime number). This seems to verify the assertion. But for $n = 40$ suddenly we have $f(40) = 41^2$: That is *not* a prime number!

What we need is a mathematical proof, verifying the correctness rigorously.

It is historically interesting to note that Euclid did *not* prove the correctness of his algorithm! He in fact verified the result of the algorithm only for one or three loops. Not having the notion of a proof by mathematical induction, he could only give a proof for a finite number of cases. (In fact he often proved only the case $n = 3$ of a theorem he wanted to establish for general n.) Although Euclid is justly famous for the great advantages he made in the art of logical deduction, techniques for giving valid proofs were not discovered until many centuries later. The crucial ideas for proving the validity of *algorithms* are only nowadays becoming really clear [29] p.336.

2.1.1 Correctness of Euclid's algorithm

Definition 2.1. A *common divisor* of two integers m and n is an integer that divides both m and n. □

We have so far tacitly supposed that there always exists a greatest common divisor. To be rigorous we have to show two things: There exists at least one divisor, and there are finitely many divisors. But we already know that 1 is always a divisor; on the other hand the set of all divisors is finite, because by Theorem A.3 (iv) all divisors have an absolute value bounded by $|n|$, as long as $n \neq 0$. Thus there are at most $2n - 1$ divisors of a non-vanishing n. In a finite

non-empty set there is an upper bound, the unique *greatest common divisor* of m,n, denoted $\gcd(m,n)$. By our short discussion we can conclude

$$1 \leqq \gcd(m,n) \leqq \max(|m|,|n|) \qquad \text{if } m \neq 0 \text{ or } n \neq 0. \tag{2.2}$$

For completeness we define gcd(0,0)=0. Hence

$$0 \leqq \gcd(m,n) \leqq \max(|m|,|n|) \qquad \forall m,n \in \mathbb{Z}. \tag{2.3}$$

Theorem 2.2. *For $m,n \in \mathbb{Z}$ we have the following:*
(i) $gcd(m,0) = |m|$.
(ii) If $n \neq 0$, then $gcd(m,n) = gcd(|n|, m \bmod n)$.

Proof. The first assertion (i) is obvious. We prove the second assertion. By Theorem A.4, there is an integer q with

$$m = q|n| + (m \bmod |n|).$$

Therefore, $\gcd(m,n)$ divides $\gcd(|n|, m \bmod |n|)$ and vice versa. Since both common divisors are nonnegative, the assertion follows from Theorem A.3 (v). Q.E.D.

Now we are ready to prove the correctness of Euclid's algorithm:

Theorem 2.3. *Euclid's algorithm computes the greatest common divisor of m and n.*

Proof. To prove that the algorithm terminates and yields $\gcd(m,n)$ we introduce some notation that will also be used later. We set

$$r_0 = |m|, \quad r_1 = |n| \tag{2.4}$$

and for $k \geqq 1$ and $r_k \neq 0$

$$r_{k+1} = r_{k-1} \bmod r_k. \tag{2.5}$$

Then $r_2, r_3, \ldots$ is the sequence of remainders that are computed in the *while*-loop. Also, after the k-th iteration of the *while*-loop we have

$$m \leftarrow r_{k+1}, \quad n \leftarrow r_k.$$

It follows from Theorem 2.2 (ii) that $\gcd(r_{k+1}, r_k) = \gcd(m,n)$ is not changed during the algorithm, as long as $r_{k+1} > 0$. Thus we only need to prove that there *is* a k such that $r_k = 0$. But this follows from the fact that by (2.5) the sequence $(r_k)_{k \geqq 1}$ is strictly decreasing, so the algorithm terminates surely. But if $r_{k+1} = 0$, we have simply that $\gcd(r_{k-1}, r_k) = r_k$, and thus $n = r_k$ is the correct result.

This concludes the proof of the correctness of the Euclidean algorithm, since after a finite time it yields the $\gcd(m,n)$. Q.E.D.

2.2 Complexity to measure efficiency

Another important aspect of algorithmic analysis is the *complexity* of an algorithm. There are two kinds of complexity which are relevant for an algorithm: Time complexity $T(n)$ and space complexity $S(n)$. The time complexity is measured by the running time it requires from the start until its termination, and the space complexity measures its required memory space when implemented on a computer.

To analyze the running time and the space requirement of an algorithm exactly, we must know the details about the implementation technology, such as hardware and software. For instance, the running time of a given algorithm depends on the frequency of the CPU, and also on the underlying computer architecture; the required memory space, on the other hands, depends on the programming language and its representation of data structures. To determine the complexities of an algorithm thus appears as an impracticable task. Moreover, the running time and required space calculated in this way are not only properties of the considered algorithm, but also of the implementation technology. However, we would appreciate some measures which are independent from the implementation technology. To obtain such asymptotic and "robust" measures, the O-notation has been introduced.

2.2.1 Asymptotic notation and complexity classes

The notations we use to describe the complexities of an algorithm are defined in terms of functions

$$T : \mathbb{N} \to \mathbb{R}^+, \qquad n \mapsto T(n),$$

where $\mathbb{R}^+$ denotes the set of nonnegative real numbers

$$\mathbb{R}^+ = \{x \in \mathbb{R} |\ x \geqq 0\} = [0, \infty). \tag{2.6}$$

That means, the domain of T consists of the natural numbers, which are mapped to a nonnegative real number. For example,

$$T(n) = 2n^2 + n + 1, \qquad \text{or} \qquad T(n) = n \ln n.$$

The complexity class $O(g(n))$

The *complexity class* $O(g(n))$ of a function $g : \mathbb{N} \to \mathbb{R}^+$ is defined as the set of all functions $f(n)$ which are dominated by $g(n)$: We say that a function $f(n)$ is *dominated* by $g(n)$ if there exist two constants $c \in \mathbb{R}^+$, $n_0 \in \mathbb{N}$ such that $f(n) \leqq cg(n)\ \forall n \geqq n_0$. That is, $f(n)$ is smaller than a constant multiple of $g(n)$ for all n greater than some finite value n_0. In other words,

$$\boxed{f(n) \in O(g(n))} \quad \text{if } \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ such that } f(n) \leqq cg(n)\ \forall n \geqq n_0. \tag{2.7}$$

O is also referred to as a *Landau symbol* or the "big-O". Figure 2.1 (a) illustrates the O-symbol. Although $O(g(n))$ denotes a *set* of functions $f(n)$ having the property (2.7), it is common to

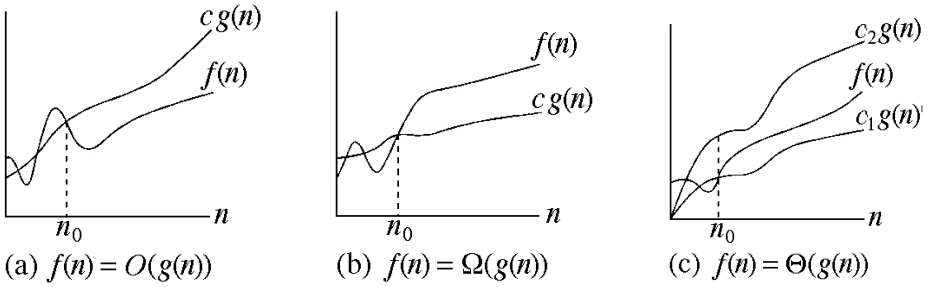


Figure 2.1: Graphic examples of the O, Ω, and Θ notations. In each part, the value of n_0 is shown as the minimum possible value; of course, any greater value would also work. (a) O-notation gives an upper bound for a function up to a constant factor. (b) Ω-notation gives a lower bound for a function up to a constant factor. (c) Θ-notation bounds a function up to constant factors.

write $\boxed{\text{"}f(n) = O(g(n))\text{"}}$ instead. We use the big-O-notation to give an *asymptotic upper bound* on a function $f(n)$, up to a constant factor.

Example 2.4. (i) We have $2n^2+n+1 = O(n^2)$, because $2n^2+n+1 \leqq 4n^2$ for all $n \geqq 1$. (That is, $c = 4$, $n_0 = 1$ in (2.7); note that we could have chosen $c = 3$ and $n_0 = 2$).

(ii) More general, any quadratic polynomial $a_2n^2 + a_1n + a_0 = O(n^2)$. To show this we assume $c = |a_2| + |a_1| + |a_0|$; then

$$a_2n^2 + a_1n + a_0 \leqq cn^2 \qquad \forall n \geqq n_0 \text{ (with } n_0 = 1)$$

because each summand is smaller than cn^2.

(iii) *(b-adic expansion)* Let b be an integer with $b > 1$. Then any number $n \in \mathbb{N}_0$ can be represented uniquely by a finite series

$$n = \sum_{i=0}^{m} a_i b^i \qquad \text{with } a_i \in \{0, 1, \ldots, b-1\}. \tag{2.8}$$

The maximum index m depends on n.[1] We write the expansion as digits $(a_n a_{n-1} \ldots a_1 a_0)_b$. Some examples:

$$\begin{aligned}
b = 2: &\quad 25 = 1\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = (11001)_2 \\
b = 3: &\quad 25 = 2\cdot 3^2 + 2\cdot 3^1 + 1\cdot 3^0 = (221)_3 \\
b = 4: &\quad 25 = 1\cdot 4^2 + 2\cdot 4^1 + 1\cdot 4^0 = (121)_4 \\
b = 5: &\quad 25 = 1\cdot 5^2 + 0\cdot 5^1 + 0\cdot 5^0 = (100)_5
\end{aligned}$$

Let now denote $l_b(n)$ the length of the b-adic expansion of a positive integer n. Then

$$l_b(n) = \lfloor \log_b n \rfloor + 1 \leqq \log_b n + 1 = \frac{\ln n}{\ln b} + 1.$$

If $n \geqq 3$ (i.e., $n_0 = 3$), we have $\ln n > 1$, and therefore $\frac{\ln n}{\ln b} + 1 < \left(\frac{1}{\ln b} + 1\right) \ln n$, i.e.,

$$l(n) \;<\; c \ln n \qquad \text{for } n \geqq 3 \text{ and with } c = \frac{1}{\ln b} + 1.$$

Therefore we have

$$l_b(n) = O(\ln n), \tag{2.9}$$

no matter what the value of b is. Therefore the number of digits of n in any number system belongs to the same complexity class $O(\ln n)$. □

The complexity class $\Omega(g(n))$

The Ω-notation provides an *asymptotic lower bound*. For two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ we write

$$\boxed{\text{“}f(n) = \Omega(g(n))\text{”} \Leftrightarrow f(n) \in \Omega(g(n))} \quad \text{if } \exists\, c \in \mathbb{R}^+,\, n_0 \in \mathbb{N} \text{ such that } cg(n) \leqq f(n)\ \forall n \geqq n_0. \tag{2.10}$$

We say that now $f(n)$ dominates $g(n)$. The intuition behind Ω is shown in figure 2.1 (b).

Example 2.5. We have $\frac{1}{2}n^3 - n + 1 = \Omega(n^2)$, because $\frac{1}{2}n^3 - n + 1 > \frac{1}{3}n^2$ for all $n \geqq 1$. (That is, $c = \frac{1}{3}$, $n_0 = 1$ in (2.10)). □

[1] This is an important result from elementary number theory. It is proved in any basic mathematical textbook, e.g. [13, 34].

The complexity class $\Theta(g(n))$

If a function $f(n)$ satisfies both conditions $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, we call it *asymptotically tightly bounded* by $g(n)$), and we write

$$\boxed{\text{“}f(n) = \Theta(g(n))\text{.”}} \qquad \text{or (correctly):} \qquad \boxed{f(n) \in \Theta(g(n)).} \tag{2.11}$$

A function $f(n)$ thus belongs to the set $\Theta(g(n))$ if there are two positive constants c_1 and c_2 such that it can be “sandwiched” between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large n. Figure 2.1 (c) gives an intuitive picture of the functions $f(n)$ and $g(n)$. For all values of n right of n_0 $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geqq n_0$ the function $f(n)$ is equal to the function $g(n)$ up to a constant factor.

The definition of $\Theta(g(n))$ requires that every member $f(n)$ of $\Theta(g(n))$ is *asymptotically nonnegative*, i.e. $f(n) \geqq 0$ whenever n is sufficiently large. Consequently, the function $g(n)$ itself must be asymptotically nonnegative (or else $\Theta(g(n))$ is empty).

Example 2.6. (i) Since we have $2n^2+n+1 = O(n^2)$ and $2n^2+n+1 = \Omega(n^2)$, we also have $2n^2+n+1 = \Theta(n^2)$.

(ii) Let b be an integer with $b > 1$ nad $l_b(n) = \lfloor \log_b n \rfloor + 1$ the length of the b-adic expansion of a positive integer n. Then $(c-1)\ln n \ \leqq \ l_b(n) < c \ln n \qquad$ for $n \geqq 3$ and with $c = \frac{1}{\ln b} + 1$. Therefore we have

$$l_b(n) \ = \ \Theta(\ln n). \tag{2.12}$$

□

The complexity classes of polynomials are rather easy to determine. A *polynomial* $f_k(n)$ of degree of degree k for some $k \in \mathbb{N}_0$ is the sum

$$f_k(n) = \sum_{i=0}^{k} a_i n^i = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_k n^k,$$

with the constant coefficients $a_i \in \mathbb{R}$. We can then state the following theorem.

Theorem 2.7. *A polynomial of degree k is contained in the complexity class $\Theta(n^k)$, i.e., $f_k(n) = \Theta(n^k)$.*

Example 2.8. We saw above that the polynomial $2n^2+n+1$ is in the complexity class $\Theta(n^2)$, according to the theorem. The polynomial, however, is not contained in the following complexity classes:

$$2n^2+n+1 \neq O(n), \qquad 2n^2+n+1 \neq \Omega(n^3), \qquad 2n^2+n+1 \neq \Theta(n^3);$$

but $2n^2+n+1 = O(n^3)$. □

2.2.2 Time complexity

The *running time* $T(n)$ of an algorithm on a particular input of size n is the number of instructions (“steps”) executed. We roughly assume a constant amount t_0 of time for each instruction. We can therefore restrict ourselves to only “counting” the steps executed doing the algorithm, because the real physical time then is achieved by multiplying with t_0. For example, if in an algorithm for input of size n the number of instructions executed is $2n^2+3$, then we will write for short

$$T(n) = 2n^2 + 3,$$

although we should write $T(n) = (2n^2+3) \cdot t_0$. This is common in computer science, because t_0 is a quantity that is machine-dependent and does not depend from the algorithm.

Analysis of the running time T is done in two ways:

1. *Worst-case analysis* determines the upper bound of running time for any input. Knowing it will give us the guarantee that the algorithm will never take any longer.

2. *Average-case analysis* determines the running time of a typical input, i.e. the expected running time. It sometimes may come out that the average time is as bad as the worst-case running time.

The complexity of an algorithm is measured in the number $T(n)$ of instructions to be done, where T is a function depending on the size of the input data n. If, e.g., $T(n) = 3n+4$, we say that the algorithm "is of linear time complexity," because $T(n) = 3n+4$ is a linear function. Time complexity functions that occur frequently are given in the following table, cf. Figure 2.2.

Complexity $T(n)$	Notation
$\ln n$, $\log_2 n$, $\log_{10} n$, ...	logarithmic time complexity $\Theta(\log n)$
n, n^2, n^3, ...	polynomial time complexity $\Theta(n^k)$
2^n, e^n, 3^n, 10^n, ...	exponential time complexity $\Theta(k^n)$

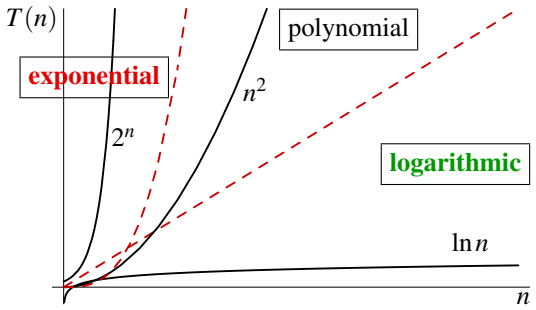


Figure 2.2: Qualitative behavior of typical functions of the three complexity classes $O(\ln n)$, $O(n^k)$, $O(k^n)$, $k \in \mathbb{R}^+$.

Definition 2.9. An algorithm is called *efficient*, if $T(n) = O(n^k)$ for a constant k, i.e., if it has polynomial time complexity or is even logarithmic. □

Analyzing even a simple algorithm can be a serious challenge. The mathematical tools required include discrete combinatorics, probability theory, algebraic skill, and the ability to identify the most significant terms in a formula.

Example 2.10. It can be proved[2] that the Euclidean algorithm has a running time

$$T_{\text{Euclid}}(m,n) = O(\log^2(mn)), \tag{2.13}$$

if all divisions and iterative steps are considered. (However, it may terminate even for large numbers m and n after a single iteration step, namely if $m \mid n$ or $n \mid m$.) Therefore, the Euclidean algorithm is efficient, since it has logarithmic running time in the worst case, in dependence of the sizes of its input numbers. □

2.2.3 Algorithmic analysis of some control structures

In the sequel let S_1 and S_2 be instructions (or instruction blocks) with running times $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$. We assume that both $f(n)$ and $g(n)$ differ from zero, i.e. $O(f(n))$ is at least $O(1)$:

$$O(1) \subseteq O(f(n)), \qquad O(1) \subseteq O(g(n)).$$

[2] Cf. [5, p902]; for the number of iterations we have $T_{\text{Euclid}}(m,n) = \Theta(\log \max[m,n])$, see footnote 4 on p. 15

- The running time of an operation is $O(1)$. A sequence of c operations is $c \cdot O(1) = O(1)$.
- A sequence $S_1; S_2$ has running time

$$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Usually, one of the functions $f(n)$ or $g(n)$ is dominant, that is $f(n) = O(g(n))$ or $g(n) = O(f(n))$. Then we have

$$T(n) = \begin{cases} O(f(n)) & \text{if } g(n) = O(f(n)), \\ O(g(n)) & \text{if } f(n) = O(g(n)). \end{cases} \tag{2.14}$$

The running time of a sequence of instructions can thus be estimated by the running time of the worst instruction.

- A selection

if (C)
  S_1
else
  S_2

consists of the condition C (an operation) and the instructions S_1 and S_2. It thus has running time $T(n) = O(1) + O(f(n)) + O(g(n))$, i.e.

$$T(n) = \begin{cases} O(f(n)) & \text{if } g(n) = O(f(n)), \\ O(g(n)) & \text{if } f(n) = O(g(n)). \end{cases} \tag{2.15}$$

- In a repetition each loop can have a different running time. All these running times have to be summed up. Let be $f(n)$ the number of loops to be done, and $g(n)$ be the running time of one loop. (Note that $f(n) = O(1)$ if the number of loops does not depend on n) Then the total running time $T(n)$ of the repetition is given by $T(n) = O(f(n)) \cdot O(g(n))$, or

$$T(n) = O\big(f(n) \cdot g(n)\big). \tag{2.16}$$

The same properties hold true for Ω and Θ, respectively.

Example 2.11. Let us examine the time complexities of the operations `search`, `insert`, and `delete` in some data structures of n nodes have. To find a particular node in a linked list, for instance, we have to start at the head of the list and — in the worst case that the last one is the searched node — run through the whole list. That is, the worst case implies n comparisons. Let a comparison an a computer take time c; this is a constant, independent from the magnitude n of the list, but depending on the machine (e.g., on speed of the processor, on quality of the compiler). So the worst-case total running time is $T(n) = c \cdot n$. In O-notation this means

$$T(n) = O(n).$$

After a node is found, deleting it requires a constant amount of running time, namely setting two pointers (for a doubly linked list: four pointers). Therefore, we have for the deletion method

$$T(n) = O(1).$$

Similarly, $T(n) = O(n)$ for the insertion of a node. To sum up, the running times for linked lists and other data structures are given in Table 2.1. □

Data structure	**search**	**insert**	**delete**
linked list	$O(n)$	$O(1)$	$O(1)$
array	$O(n)$	$O(n)$	$O(n)$
sorted array	$O(\ln n)$	$O(n)$	$O(n)$

Table 2.1: Running times $T(n)$ for operations on data structures of n nodes in the worst cases.

2.3 Summary

- Algorithmic analysis proves the correctness and studies the complexity of an algorithm by mathematical means. The complexity is measured by counting the number of instructions that have to be done during the algorithm on a RAM, an idealized mathematical model of a computer.

- Asymptotic notation erases the "fine structure" of a function and lets only survive its asymptotic behavior for large numbers. The O, Ω, and Θ-notation provide an asymptotical bounds on a function. We use them to simplify complexity analysis. If the running time of an algorithm with input size n is $T(n) = 5n+2$ we may say simply that it is $O(n)$. The following essential aspects have to be kept in mind:

  – The O-notation eliminates constants: $O(n) = O(n/2) = O(17n) = O(6n+5)$. For all these expressions we write $O(n)$. The same holds true for the Ω-notation and the Θ-notation.

  – The O-notation yields upper bounds: $O(1) \subset O(n) \subset O(n^2) \subset O(2^n)$. (Note that you cannot change the sequence of relations!) So it is not wrong to say $3n^2 = O(n^5)$.

  – The Ω-notation yields lower bounds: $\Omega(2^n) \subset \Omega(n^2) \subset \Omega(n) \subset \Omega(1)$. So, $3n^5 = \Omega(n^3)$.

  – The Θ-notation yields tight bounds: $\Theta(1) \not\subset \Theta(n) \not\subset \Theta(n^2) \not\subset \Theta(2^n)$. So $3n^2 = \Theta(n^2)$, but $3n^2 \neq \Theta(n^5)$.

- Suggestively, the notations correspond to the signs $\leqq$, $=$, and $\geqq$ as follows:

$T(n) = O(g(n))$	"$T(n) \leqq g(n)$"
$T(n) = \Theta(g(n))$	"$T(n) = g(n)$"
$T(n) = \Omega(g(n))$	"$T(n) \geqq g(n)$"

- The O-notation simplifies the worst-case analysis of algorithms, the Θ-notation is used if exact complexity classes can be determined. For many algorithms, a tight complexity bound is not possible! For instance, the termination of the Euclidean algorithm does not only depend on the size of m and n, even for giant numbers such as $m = 10^{100^{100}}$ and $n = 10^{10^{99}}$ it may terminate after a single step: $\gcd(m,n) = n$.

- There are three essential classes of complexity, the class of logarithmic functions $O(\log n)$, of polynomial functions $O(n^k)$, and of exponential functions $O(k^n)$, for any $k \in \mathbb{R}^+$.

- An algorithm with polynomial time complexity is called efficient.

Chapter 3

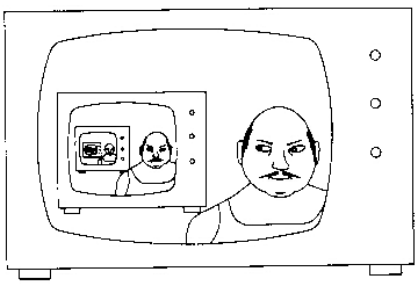
Recursions

3.1 Introduction

Building stacks is closely related to the phenomenon of *recursions*. Building stacks again are related to the construction of relative phrases in human languages. In the most extreme form recursions in human language probably occur in German: the notorious property of the German language to put the verb at the end of a relative phrase has a classical persiflage due to Christian Morgenstern at the beginning of his *Galgenlieder*:

Es darf daher getrost,
was auch von allen,
deren Sinne,
weil sie unter Sternen,
die,
wie der Dichter sagt,
zu dörren, statt zu leuchten, geschaffen sind
geboren sind,
vertrocknet sind,
behauptet wird,
enthauptet werden ...

A case of recursion is shown in figure 3.1. Such a phenomenon is referred to as "feedback" in the



(‡ Restricted Use)

Figure 3.1: Recursion. Figure taken from [Wirth (1999)]

engineerings. Everyone knows the effect of a microphone held near a loudspeaker amplifying the input of this microphone ... the high whistling noise is unforgetable.

3.2 Recursive algorithms

Recursion is a fundamental concept as well as in mathematics as in computer science. A *recursive algorithm* is an algorithm which calls itself. Of course, a recursive algorithm cannot always call itself, for if so, it would be circular. Another important property of a valid recursive algorithm is the stopping condition, which gives a definite end to the calling sequence.

Let us first look at a mathematical example. The factorial of a nonnegative integer n is written as $n!$ (pronounced "n factorial"). It is defined as the product

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1.$$

So $1! = 1$, $3! = 3 \cdot 2 \cdot 1 = 6$, and $5! = 120$. (By definition, 0!=1.) An algorithm $\mathrm{fac}(n)$ determining $n!$ may be defined as follows:

```
algorithm fac (n)
if ( i == 0 )
    return 1;
else
    return n · fac (n-1);
```

For any $n \in \mathbb{Z}$ we obtain $n! = \mathrm{fac}(n)$. Why? Now, let us prove it by induction:

- *Induction start.* For $n = 0$ we have $\mathrm{fac}(0) = 1$. Hence $\mathrm{fac}(0) = 0!$.
- *Induction step* $n \to n+1$. We assume that

$$\mathrm{fac}(n) = n! \tag{3.1}$$

Thus we may conclude

$$\mathrm{fac}(n+1) \underset{\text{by def.}}{=} (n+1) \cdot \mathrm{fac}(n) \underset{(3.1)}{=} (n+1) \cdot n! = (n+1)!$$

O.k., you perhaps believe this proof, but maybe you do not see why this recursion works? Consider for example the case $n = 3$:

call $\mathrm{fac}(3)$ which yields $\mathrm{fac}(3) = 3 \cdot \boxed{\mathrm{fac}(2)}$

call $\boxed{\mathrm{fac}(2)}$ which yields $\mathrm{fac}(2) = 2 \cdot \mathrm{fac}(1)$

call $\mathrm{fac}(1)$ which yields $\mathrm{fac}(1) = 1 \cdot \mathrm{fac}(0)$

call $\mathrm{fac}(0)$ which returns $\boxed{1}$

this yields $\mathrm{fac}(1) = 1 \cdot \boxed{1} = 1$, hence returns 1

this yields $\mathrm{fac}(2) = 2 \cdot 1 = 2$, hence returns 2

this yields $\mathrm{fac}(3) = 3 \cdot 2 = 6$, hence returns 6

Voilà, $\mathrm{fac}(3)$ = 6 is the result! The call and return sequence is shown in figure 3.2. We can make the following general observations. A recursive algorithm (here $\mathrm{fac}(n)$) is called to solve a problem. The algorithm actually "knows" how to solve only the simplest case or so called *base case* (or base cases, here $n = 0$). If the algorithm is called with this base case, it returns a result. If the algorithm is called with a more complex problem, it divides the problem into

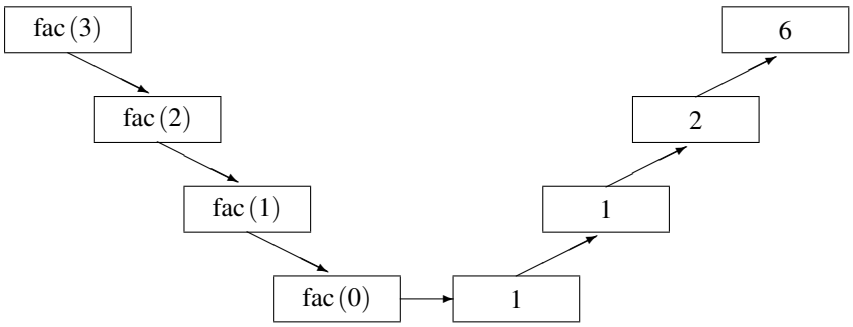


Figure 3.2: The call and return sequence of factorial.

two pieces: one piece that the algorithm knows how to do (base case), and one piece that it does not know. The latter piece must resemble the original problem, but be a slightly simpler or smaller version of the original problem. Because this new problem looks like the original one, the algorithm calls itself to go to work with the smaller problem — this is referred to as a *recursive call* or the *recursion step*.

The recursion step executes while the original call of the algorithm is still open (i.e., it has not finished executing). The recursion step can result in many more recursive calls, as the algorithm divides each new subproblem into two conceptual pieces. For the recursion to eventually terminate, each time the algorithm calls itself with a smaller version of the problem, the sequence of smaller and smaller problems must converge to the base case in finite time. At that point the algorithm recognizes the base case, returns a result to the previous algorithm and a sequence of returns ensues up the line until the first algorithm returns the final result.

Recursion resembles much the concept of mathematical induction, which we learned above. In fact, the problem $P(n)$ is proven by reducing it to be true if the smaller problem $P(n-1)$ is true. This in turn is true if the smaller problem $P(n-2)$ is true, and so on. Finally, the base case, called induction start, is reached which is proven to be true.

3.3 Searching the maximum in an array

Let us now look at another example, finding the maximum element on an array $a[]$ of integers. The strategy is to split the array in two halves and take the half whose maximum is greater than the maximum of the other one, until we reach the base cases where there remains only one or two nodes. Let be l, r two integers. Then the algorithm *maximum* $(a[],l,r)$ is defined by:

```
algorithm searchmax(a[],l,r) // — find the maximum a[l],a[l+1],...,a[r]
if (l = r)                                   // the base case
    return a[l];
else m ← searchmax(a[],l+1,r)                // remains open until base case is reached!
    if ( a[l] > m )
      return a[l]
    else
      return m;
```

The solution can be described by the illustration in figure 3.3. Another way to visualize the working of searchmax (and a general recursive algorithm as well) is figure 3.4. It shows the sequence of successive calls and respective returns.

The searchmax algorithm for an array of length n takes exactly $2n$ operations, namely n calls

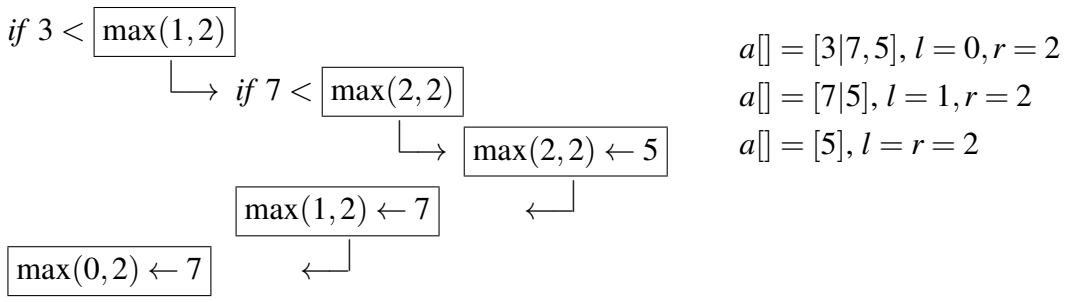


Figure 3.3: The searchmax algorithm.

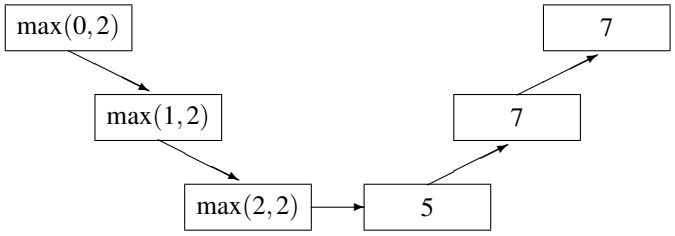


Figure 3.4: The call and return sequence searchmax.

and n returns. Thus the running time of this algorithm is

$$T_{\text{searchmax}}(n) = O(n). \tag{3.2}$$

Exercise 3.1. Try this algorithm out with the input array

$$a[] = [3,9,2,8,6], \quad l = 0 \quad r = 4.$$

3.4 Recursion versus iteration

In this section we compare the two approaches of recursion and iteration and discuss why one might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure (*while*); recursion uses a selection structure (*if*). Both iteration and recursion involve repetition: Iteration explicitly uses the repetition structure; recursion achieves repetition through repeated subroutin (method or function) calls. Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Both iteration and recursion can occur infintely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges to the base case.

Practically, recursion has many negatives. It repeatedly invokes the mechanism, and consequently the overhead, of method calls. This can be expensive in both processor time and memory space. Each recursive call causes another copy of the method (actually, only the method's variables!) to be created. Iteration normally occurs *within* a method, so the overhead of repeated method calls and extra memory assignment is omitted. So why recursion?

Rule 1. *Any recursion consisting of a single recursion call in each step (a "primitive recursion") can be implemented as an iteration, and vice versa.*

As an example for the fact that any recursive algorithm can be substituted by an iterative one let us look at the following iterative definition of fac (n) determining the value of $n!$:

algorithm **fac2** (n)
$f \leftarrow 1;$
for $(\ i = 1;\ i \leqq n;\ i{+}{+}\ )$
    $f \leftarrow i \cdot f;$
return f;

A recursive approach is chosen in preference to an iterative one when it more naturally mirrors the problem and results in an algorithm that is easier to understand. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

3.4.1 Recursive extended Euclidean algorithm

There is a short recursive version of the extended Euclidean algorithm which computes additional useful information. Specifically, the algorithm invoked with the integers m and n computes the integer coefficients x_0, x_1, x_2 such that

$$x_0 = \gcd(m,n) = x_1 m + x_2 n. \tag{3.3}$$

Note that x_1 and x_2 may be zero or negative. These coefficients are very useful for the solution of linear Diophantine equations, particularly for the computation of modular multiplicative inverses in cryptology. The following algorithm `extendedEuclid` takes as input an arbitrary pair (m,n) of positive integers and returns a triple of the form (x_0, x_1, x_2) that satisfies Equation (3.3).

```
algorithm extendedEuclid( long m, long n ) {
   long[] x = {m, 1, 0};
   long tmp;

   if ( n == 0 ) {
      return x;
   } else {
      x = extendedEuclid( n, m % n );
      tmp = x[1];
      x[1] = x[2];
      x[2] = tmp - (m/n) * x[2];
      return x;
   }
}
```

3.5 Complexity of recursive algorithms

In general, analyzing the complexity of recursive algorithm is not a trivial problem. We will first compute the running time of the recursive factorial algorithm to outline the principles.

Look at the left part of figure 3.5. Here we see the sequence of recursive calls of $fac\,(n)$, starting at the top with the call of $fac\,(n)$, followed by the call of $fac\,(n-1)$ and so on, until we reach the base case $n = 0$. This yields a so-called *recursion tree*. (It is a rather simple tree, with only one branch at each generation.)

At the base case, we achieve the solution $fac\,(0) = 1$ which is returned to the previous generation, etc. We count that the recursion tree has $n+1$ levels (or generations).

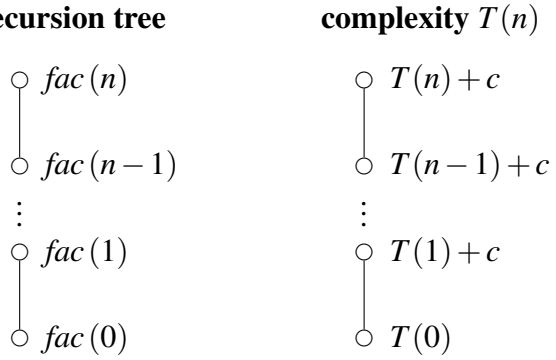


Figure 3.5: Recursion tree of calls of the factorial algorithm and the respective running times $T(n)$.

If we want to analyze the complexity, we have to compute the running time on each level. Let be $n = 0$. Then the running time $T(0)$ is a constant c_0,

$$T(0) = c_0.$$

c_0 is given by the operations

- determining whether $n = 0$ (comparison)
- deciding to execute the base case (if-statement)
- assigning (or returning) the value 1.

So we could for instance estimate $c_0 = 3$ (Remember that this is a rough estimate! It depends on the concrete computer machine how much elementary operations indeed are executed for an arithmetic operation or an assignment.)

What then about $T(1)$? We first see that there are the following operations to be done:

- determining whether $n = 0$ (comparison)
- deciding to execute the else case (if-statement)
- calling $fac(0)$.

This results in a running time

$$T(1) = T(0) + c,$$

where c is a constant (which is approximately 3: $c \approx 3$). But we already know $T(0)$, and so we have $T(1) = c + c_0$. Now, analogously to the induction step we can conclude: The running time $T(n)$ for any $n \geqq 1$ is given by

$$T(n) = T(n-1) + c.$$

To summarize, we therefore achieve the following equation for the running time $T(n)$ for an arbitrary $n \geqq 0$:

$$T(n) = \begin{cases} c_0 & \text{if } n = 0, \\ T(n-1) + c & \text{if } n \geqq 1. \end{cases} \tag{3.4}$$

This is a *recurrence equation*, or short: a *recurrence*. It is an equation which not directly yields a closed formula for $T(n)$ but which yields a *construction plan to calculate* $T(n)$. In fact, for this special recurrence equation we achieve the simple solution $T(n) = n \cdot c + c_0$, i.e.,

$$\boxed{T(n) = \Theta(n).} \tag{3.5}$$

For wide class of recursion algorithms, the time complexity can be estimated by the so-called *Master theorem* [5, §4.3]. A simple version of it is the following theorem.

Theorem 3.2 (**Master Theorem, special case**). *Let $a \geqq 0$, $b > 1$ be constants, and let $T : \mathbb{N} \to \mathbb{R}^+$ be a function defined by the recurrence*

$$T(n) = \begin{cases} T_0 & \text{if } n = n_0, \\ a\,T\left(\lfloor n/b \rfloor\right) + \Theta(n^{\log_b a}) & \text{otherwise} \end{cases} \tag{3.6}$$

with some initial value T_0. Then the $T(n)$ can be estimated asymptotically as being polynomial:

$$\boxed{T(n) = \Theta(n^{\log_b a} \log n).} \tag{3.7}$$

The same property holds true if $T(\lfloor \frac{n}{b} \rfloor)$ is replaced by $T(\lceil \frac{n}{b} \rceil)$.

Proof. A much more general case is proved in [5, §4.4]. In fact, even this result is a special case of the Akra-Bazzi Theorem published in 1998. It covers a very wide range of recursion equations. □

The following theorem is the mathematical basis of the asymptotic estimation of the "divide-and-conquer" algorithms, an important construction principle which we will see later in section 4.3. The theorem gives an asymtotic approximation of a special type of *recurrence equations* using a simple property of binary trees, cf. Eq. (3.4) p. 29).

Theorem 3.3 (**Divide-and-Conquer Theorem**). *Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ be a function which grows at least linearly in n, i.e. which satisfies*

$$a \cdot f\left(\tfrac{n}{a}\right) \leqq f(n) \qquad \text{for } a \in \mathbb{N}. \tag{3.8}$$

A function $T : \mathbb{N} \to \mathbb{R}^+$ obeying the recursion equation

$$T(n) = \begin{cases} f(1) & \text{if } n = 1, \\ 2\,T\left(n/2\right) + f(n) & \text{if } n > 1. \end{cases} \tag{3.9}$$

then can be asymptotically estimated as

$$\boxed{T(n) = O(f(n) \log n).} \tag{3.10}$$

Especially for a linear function satisfyig $af(\frac{n}{a}) = f(n)$ instead of (3.8) we have

$$\boxed{T(n) = \Theta(f(n) \log n).} \tag{3.11}$$

(The same assertions hold true if $\lfloor n/2 \rfloor$ is replaced by $\lceil n/2 \rceil$.)

Proof. However, a sketch of a proof reads as follows. Consider the tree of the possible recursive calls. It is a binary tree (representable as a left-complete tree) with $k = \lceil \log_2 n \rceil$ levels ("generations"), see figure 3.6. Because f grows at least linearly, equation (3.8), we estimate for each level:

$$\begin{array}{rrl} \text{level } 0: & & f(n) \\ 1: & 2 \cdot f(n/2) \leqq & f(n) \\ 2: & 4 \cdot f(n/4) \leqq & f(n) \\ \vdots & \vdots & \vdots \\ (k-1): & n/2 \cdot f(2) \leqq & f(n) \\ k: & n \cdot f(1) \leqq & f(n) \\ \hline & \text{sum} \leqq & (k+1) \cdot f(n) \\ & = & (1 + \lceil \log_2 n \rceil) f(n) \end{array}$$

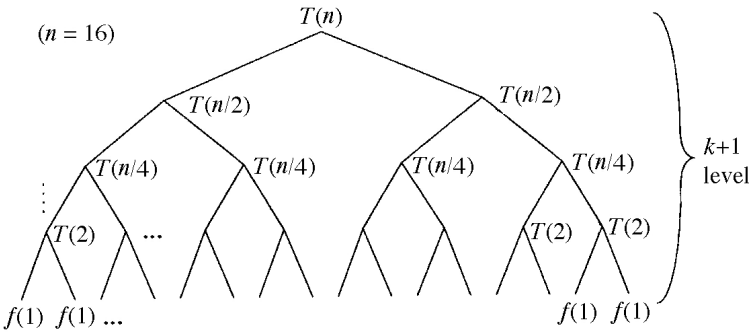


Figure 3.6: A typical call sequence of a divide and conquer algorithm

Therefore, $T(n) \leqq f(n)\,(2+\log_2 n) = O(f(n)\cdot \log n)$. If especially $af(n/a) = f(n)$, we even have $T(n) = f(n)\cdot(1+\lceil \log_2 n\rceil) = \Theta(f(n)\log n)$. □

Finally we state a result which demonstrates the power as well as the danger of recursion. It is quite easy to generate exponential growth.

Theorem 3.4. *Let $a \geqq 2$ be an integer and $f : \mathbb{R}^+ \to \mathbb{R}^+$ a positive function of at most polynomial growth, i.e., there exists a constant $k \in \mathbb{N}_0$ such that $f(n) = O(n^k)$. Then a function $T : \mathbb{N} \to \mathbb{R}^+$ obeying the recursion equation*

$$T(n) = \begin{cases} f(0) & \text{if } n = 0, \\ aT(n-1) + f(n) & \text{if } n > 0. \end{cases} \tag{3.12}$$

can be asymptotically estimated as

$$\boxed{T(n) = \Theta(a^n).} \tag{3.13}$$

Proof. Analogously to Fig. 3.6, we see that according to Eq. (3.12) there are n generation levels in the call tree of $T(n)$, and therefore a^n basis cases. As long as f grows at most polynomially, this means that $T(n) = \Theta(a^n)$. □

Examples 3.5. (i) The recursion equation $T(n) = \frac{1}{2}T(\frac{n}{2}) + n$ is in the class Eq. (3.6) with $a = b = 2$, hence $T(n) = \Theta(n\log n)$.

(ii) A function $T(n)$ satisfying $T(\lceil \frac{n}{2}\rceil) + 1$ is of the class (3.9) with $f(n) = 1$, i.e., $T(n) = O(\log n)$.

(iii) The algorithm drawing the "Koch snowflake curve", a special recursive curve, to the level n has a time complexity $T(n)$ given by $T(n) = 4T(n-1) + c_1$ with a constant c_1. Since it therefore obeys (3.12) with $a = 4$ and $f(n) = c_1$, we have $T(n) = \Theta(4^n)$. □

3.6 The towers of Hanoi

Legend has it that in a temple in the Far East, priests are attempting to move a stack of (big gold or stone) disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that

- exactly one disk is moved at a time and
- at no time may a larger disk be placed above a smaller disk.

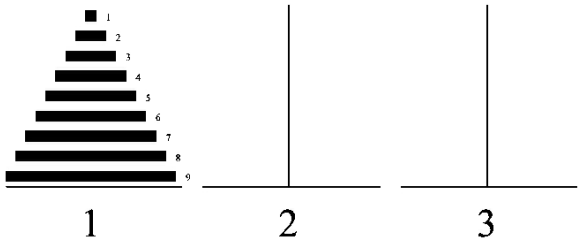


Figure 3.7: The towers of Hanoi for the case of nine disks.

A third peg is available for temporarily holding disks. So schemtaically the situation looks like as in figure 3.7. According to the legend the world will end when the priests complete their task. So we will attack the problem, but better won't tell them the solution...

Let us assume that the priests are attempting to move the disks from peg 1 to peg 2. We wish to develop an algorithm that will output the precise sequence of peg-to-peg disk transfers. For instance, the output

$$1 \to 3$$

means: "Move the most upper disk at peg 1 to the top of peg 3." For the case of only two disks, e.g., the output sequence reads

$$1 \to 3, \qquad 1 \to 2, \qquad 3 \to 1. \tag{3.14}$$

Try to solve the problem for $n = 4$ disks. There should be 15 moves.

If we were to approach the general problem with conventional methods, we would rapidly found ourselves hopelessly knotted up in managing disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving n disks can be viewed in

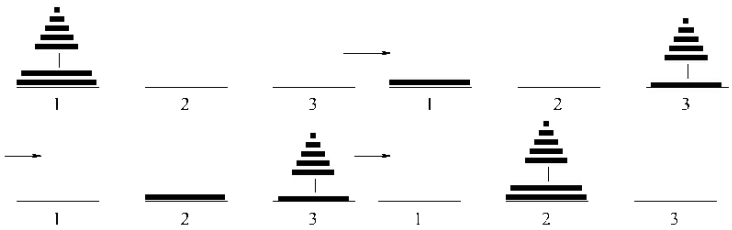


Figure 3.8: The recursive solution of the towers of Hanoi.

terms of moving only $n-1$ disks (hence the recursion) as follows.

1. Move $n-1$ disks from peg 1 to peg 3, using peg 2 as a temporary holding area.
2. Move the last disk (the largest) from peg 1 to peg 2.
3. Move the $n-1$ disks from peg 3 to peg 2, using peg 1 as a holding area.

The process ends when the last task involves moving $n = 1$ disks, i.e. the base case. This is solved trivially by moving the disk from peg 1 to peg 2, without the need of a temporarily holding area.

The formulation of the algorithm reads as follows. We name it *hanoi* and call it with four parameters:

1. the number n of disks to be moved,
2. the peg j on which the disks are initially threaded,

3. the peg k to which this stack of disks is to be moved,
4. the peg l to be used as a temporary holding area.

```
algorithm hanoi(n, j, k, l)
if (n = 1)                          // base case
    output (j, " → ", k);
else
    hanoi(n − 1, j, l, k)           // hold (n − 1)-stack temporarily on peg l
    output (j, " → ", k);
    hanoi(n − 1, l, k, j)           // move (n − 1)-stack to target peg k
```

What about the running time of this algorithm? In fact, from the recursion algorithm we can directly derive the recursion equation

$$T(n) = \begin{cases} c_0 & \text{if } n = 1, \\ 2T(n-1) + c_1 & \text{otherwise,} \end{cases} \tag{3.15}$$

with c_0 being a constant representing the output effort in the basis case, and c_1 the constant effor in the recursion step. Since this equation is of the class of Theorem 3.4 with $a = 2$ and $f(n) = c_1$ for $n > 1$, $f(1) = c_0$, we have

$$T(n) = \Theta(2^n). \tag{3.16}$$

If we try to exactly count the moves to be carried out, we achieve the number $f(n)$ of moves for the problem with n disks as follows. Regarding the algorithm, we see that $f(n) = 2f(n-1)+1$ for $n \geqq 1$, with $f(0) = 0$. (Why?) It can be proved easily by induction that then

$$f(n) = 2^n - 1 \qquad \text{for } n \geqq 0. \tag{3.17}$$

Summary

- A recursion is a subroutine calling itself during its execution. It consists of a basis case (or basis cases) which do not contain a recursive call but return certain values, and of one or several recursive steps which invoke the subroutine with slightly changed parameters. A recursion terminates if for any allowed arguments the basis case is reached after finitely many steps.
- A wide and important class of recursions, the "primitive recursions" consisting of a single recursive call in the recursion step, can be equivalently implemented iteratively, i.e., with loops.
- The time complexity of a recursive algorithm is detrmined by a recursion equation which can be directly derived from the algorithm. There are the following usual classes of recursion equations

  $$T(n) = T(n-1) + c, \qquad T(n) = bT(n-1) + c, \qquad T(n) = aT(\lfloor n/b \rfloor) + \Theta(n^{\log_b a} \log n)$$

  with constants $a \geqq 1$, $b > 1$, $c > 0$ and some appropriate base cases. These have solutions with the respective asymptotic behaviors

  $$T(n) = \Theta(n), \qquad T(n) = \Theta(b^n), \qquad T(n) = \Theta(n^{\log_b a} \log n)$$

Chapter 4

Sorting

So far we have come to know the data structures of array, stack, queue, and heap. They all allow an organization of data such that elements can be added to, or deleted from. In the next few chapter we survey the computer scientist's toolbox of frequently used algorithms and discuss their efficiency.

A big part of overall CPU time is used for sorting. The purpose of sorting is not only to get the items into a right order but also to bring together what belongs together. To see for instance all transactions belonging to a specific credit card account, it is convenient to sort the data records by credit card number and then look only at the intervall containing the respective transactions.

4.1 Simple sorting algorithms

SelectionSort. The first and easiest sorting algorithm is the *selectionSort*. We assume that the data are stored in an array $a[n]$.

```
selectionSort (a[], n)
// sorts array a[0], a[1], ..., a[n-1] ascendingly
for (i = 0; i ≦ n-2; i++) {      // find minimum of a[i], ..., a[n]
    min ← i;
    for (j = i+1; j ≦ n-1; j++) {
        if (a[j] < a[min])
            min ← j;
    }
    a[i] ↔ a[min];
}
```

Essentially, for the first loop there are made $n-1$ operations (inner j-loop), for the second one $n-1$, ... Hence its running time is given by

$$T_{\text{sel}}(n) = (n-1) + (n-2) + \ldots + 1 = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

Insertion sort. Another simple method to sort an array $a[]$ is to insert an element in the right order. This is done by the *insertion sort*. To simplify this method we define the element $a[0]$ by initializing $a[0] = -\infty$. (In practice $-\infty$ means an appropriate constant.)

```
insertionSort(a[],n) // sorts array a[0],a[1],...,a[n-1] ascendingly
for (i = 1; i ≦ n; i++){
    // greater values are moved one item up
    r ← a[i];  j ← i-1;
    while ( a[j] > r ) {
          a[j+1] ← a[j];  j ← j-1;
          }
    a[j+1] ↔ r;
}
```

In the worst case the inner *while*-loop is run through just to the beginning of the array ($a[1]$). This is the case for a descending ordered initial sequence. The effort then is given by

$$T_{\text{ins}}(n) = 1+2+\ldots+n = \sum_{k=1}^{n} k = \frac{(n+1)n}{2} = O(n^2).$$

In the mean we can expect the inner loop running through half of the lower array positions. The effort then is

$$\bar{T}_{\text{s}}(n) = \frac{1}{2}(1+2+\ldots+(n-1)) = \frac{1}{2}\sum_{i=1}^{n-1} i = \frac{n(n-1)}{4} = O(n^2).$$

BubbleSort. Bubble sort, also known as exchange sort, is a simple sorting algorithm which works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e., the beginning) of the list by the swaps.

```
bubbleSort(a[]) // sorts array a[0],a[1],...,a[n-1] ascendingly
for (i = 1; i < n; i++)
    for (j = 0; j < n-i; ++j)
        if (a[j] > a[j+1]) a[j] ↔ a[j+1];
```

Its running time again is $T_{\text{bub}}(n) = O(n^2)$. Following is a slightly improved version which does not waste running time if the array is already sorted. Here the pass through the array is repeated until no swaps are needed:

```
bubbleSort(a[]) // sorts array a[0],a[1],...,a[n-1] ascendingly
do {
    swapped ← false;
    for (j = 0; j < n-1; j++)
        if (a[j] > a[j+1])
          a[j] ↔ a[j+1];  swapped ← true;
} while (swapped);
```

4.2 Theoretical minimum complexity of a sorting algorithm

Are there better sorting algorithms? The sorting algorithms considered so far are based on comparisons of keys of elements. They are therefore called *(key) comparison sort algorithms*

It is clear that they cannot be faster than $\Omega(n)$, because each element key has to be considered. $\Omega(n)$ is an absolute lower bound for key comparison algorithms. It can be proved that *any key comparison sort algorithm needs at least*

$$T_{\text{sort}}(n) = \Omega(n \log n) \tag{4.1}$$

comparisons in the worst case for a data structure of n elements [19, §6.4]. However, in special situations there exist sorting algorithms which have a better running time, noteworthy the *pigeonhole sort* sorting an array of positive integers. It is a special version of the *bucket sort* [23, §2.7].

```
pigeonholeSort (int[] a)
    // determine maximum entry of array a:
    max ← −∞;
    for (i ← 0; i < a.length; i++) if (max < a[i]) max ← a[i];
    b = new int[max+1]; // b has max+1 pigeonholes
    for (i ← 0; i < a.length; i++) b[a[i]]++; // counts the entries of pigeonhole a[i]
    // copy the pigeonhole entries back to a:
    j ← 0;
    for (i = 0; i < b.length; i++)
        for (k = 0; k < b[i]; k++)
            a[j] = i; j++;
```

It has time complexity $O(n + \max a)$ and space complexity $O(\max a)$ where $\max a$ denotes the maximum entry of the array. Hence, if $0 \leqq a[i] \leqq O(n)$, then both time and space complexity are $O(n)$.

4.3 A recursive construction strategy: Divide and conquer

The strategy to split a problem into several parts and to solve each part recursively, is called "divide and conquer." We can formulate it as follows:

if the object set is small enough
  solve the problem directly
else
  divide: split the set into several subsets (if possible, of equal size)
  conquer: solve the problem for each subset recurively
  merge: combine the solutions of the subsets to a solution of the total problem

If the several parts have approximately equal size, the algorithm is called a *balanced divide and conquer algorithm*

Theorem 4.1. *A balanced divide and conquer algorithm has running time*
(i) $O(n)$ if the divide and merge steps each only need $O(1)$ running time;
(ii) $O(n \log n)$ if the divide and merge steps each have linear running time $O(n)$.

This theorem is a powerful result, it solves the complexity analysis of a wide class of algorithms with a single hit. Its proof is based on Theorem 3.2, with $a = b = 2$.

Remark 4.2. *For a balanced divide-and-conquer algorithm, the running time $T(n)$ is given by*

a recursion equation:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ \underbrace{O(n)}_{\text{divide}} + \underbrace{2 \cdot T(n/2)}_{\text{conquer}} + \underbrace{O(1)}_{\text{merge}} & \text{if } n > 1. \end{cases} \tag{4.2}$$

Hence $f(n) = O(1) + O(n) = O(n)$, i.e. f has linear growth. Therefore, $T(n) = O(n \log n)$.

Examples of divide and conquer algorithms are mergeSort and quickSort.

4.4 Fast sorting algorithms

4.4.1 MergeSort

Let be $a = a_0 \dots a_{n-1}$ the input sequence.

```
algorithm mergeSort(l, r)
if (l < r)    // base case l = r: nothing to do!
    m ← ⌊(l + r − 1)/2⌋;
    b ← a_l ... a_m;  c ← a_{m+1} ... a_r;
    b' ← mergeSort(l, m);
    c' ← mergeSort(m + 1, r);
    merge(l, m, r);    // merges b' and c'
```

Subalgorithm *merge* is given by:

```
algorithm merge(l, m, r)
// merges two sorted parts a_l ... a_m and a_{m+1} ... a_r using a temporary sequence t
i ← l;  j ← m + 1;
for (k = l;  k ≦ r;  k++) {
    if ( (j > r) || ( (i ≦ m) && (a_i ≦ a_j) ) )
        t_k ← a_i;  i++;
    else
        t_k ← a_j;  j++;
}
for (i = l;  i ≦ r;  i++)    // copy t → a
    a_i ← t_i;
```

The algorithm `mergeSort(`l, r`)` works as follows (see Fig. 4.1):

- *divide* the element sequence $a_l, \dots, a_r$ into two (nearly) equal sequences
- *conquer* by sorting the two sequences recursively by calling mergeSort for both sequences;
- *merge* both sorted sequences.

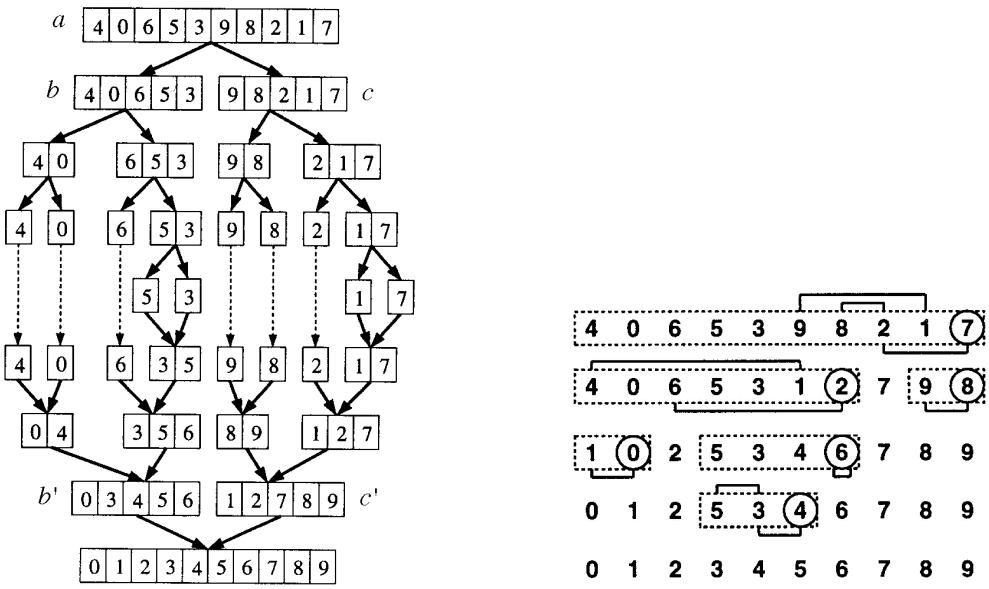


Figure 4.1: Left figure: mergeSort for an array of 10 elements. Right figure: quickSort for an array of 10 elements

4.4.2 QuickSort

This algorithm has much in common with mergeSort. It is a recursive divide and conquer algorithm as well. But whereas mergeSort uses a trivial divide step leaving the greatest part of the work to the merge step, quickSort works in the divide step and has a trivial merge step instead. Although it has a bad worst case behavior, it is probably the most used sorting algorithm. It is comparably old, developed by C.A.R. Hoare in 1962.

Let be $a = a_0 \dots a_{n-1}$ the sequence to be operated upon. The algorithm `quickSort(`l, r`)` works as follows:

- *divide* the r element sequence $a_l \dots a_r$ into two sequences $a_l, \dots, a_{p-1}$ and $a_{p+1} \dots a_r$ such that each element of the first sequence is smaller than any element of the second sequence: $a_i \leqq a_p$ with $l \leqq i < p$ and $a_j \geqq a_p$ with $p < j \leqq r$. This step we call *partition*, and the element a_p is called *pivot element*.[1] Usually, the element a_r is chosen to be the pivot element, but if you want you can choose the pivot element arbitrarily.

- *conquer* by sorting the two sequences recursively by calling quickSort for both sequences;

- there is nothing to *merge* because both sequences are sorted separately.

```
algorithm quickSort(l, r)
// for the case l ≧ r there remains nothing to do (base case)!
if (l < r)
    p ← partition(l, r);
    quicksort(l, p − 1);
    quicksort(p + 1, r);
```

Here the subalgorithm *partition* is given by

[1] *pivot*: Dreh-, Angelpunkt; Schwenkungspunkt

```
algorithm partition (l, r)
// finds the right position for the pivot element a_r
i ← l − 1; j ← r;
while ( i < j ) {
    i++;
    while (i < j && a_i < a_r)          // "i-loop"
        i++;
    j − −;
    while ( i < j && a_j > a_r )        // "j-loop"
        j − −;
    if ( i ≧ j )
        a_i ↔ a_r;
    else
        a_i ↔ a_j;
}
return i;
```

We see that after the inner "i-loop" the index i points to the first element a_i from the left which is greater than or equal a_r, $a_i \geqq a_r$ (if $i < j$). After the "j-loop" j points to the first element a_j from the right which is smaller than a_r (if $j > i$). Therefore, after the subalgorithm *partition* the pivot element a_p is placed on its right position (which will not be changed in the sequel). See Fig. 4.1.

Complexity analysis of *quickSort*

The complexity analysis of *quickSort* is not trivial. The difficulty lies in the fact that finding the pivot element a_p depends on the array. In general, this element is not in the middle of the array, and thus we do not necessarily have a balanced divide-and-conquer algorithm. :-(

The relevant step is the divide-step consisting of the partition algorithm. The outer loop is executed exactly once, whereas the two inner loops add to $n-1$. The running time for an array of length $n = 1$ is a constant c_0, and for each following step we need time c additionally to the recursion calls. Hence we achieve the recurrence equation

$$T(n) = \begin{cases} c_0 & \text{if } n = 1, \\ \underbrace{(n-1)+c}_{\text{divide}} + \underbrace{T(p-1)+T(n-p)}_{\text{conquer}} + \underbrace{0}_{\text{merge}} & \text{if } n > 1 \quad (1 \leqq p \leqq n). \end{cases} \tag{4.3}$$

Worst case

In the worst case, $p = 1$ or $p = n$ This results in a worst case recurrency equation

$$T_{\text{worst}}(n) = \begin{cases} c_0 & \text{if } n = 1, \\ (n-1) + T_{\text{worst}}(n-1) + c & \text{if } n > 1. \end{cases} \tag{4.4}$$

Building up

$$\begin{aligned} T_{\text{worst}}(1) &= c_0 \\ T_{\text{worst}}(2) &= 1+T(1)+c = 1+c+c_0 \\ T_{\text{worst}}(3) &= 2+T(2)+c = 2+1+2c+c_0 \\ &\vdots \\ T_{\text{worst}}(n) &= \sum_{k=1}^{n-1} k+(n-1)c+c_0 = \binom{n}{2}+(n-1)c+c_0 = O(n^2). \end{aligned}$$

Therefore, *quickSort* is not better than *insertionSort* in the worst case. (Unfortunately, the worst case is present, if the array is already sorted...)

Best and average case

The best case is shown easily. It means that for each recursion step the pivot index p is chosen in the middle of the array area. This means that *quickSort* then is a balanced divide-amd-conquer algorithm with a linear running time divide-step:

$$T_{\text{best}}(n) = O(n\log n). \tag{4.5}$$

It can be shown that the average case is only slightly longer [23] §2.4.3:

$$T_{\text{average}}(n) = O(n\log n). \tag{4.6}$$

4.4.3 HeapSort

Because of the efficient implementability of a heap in an array, it is of great practical interest to consider heapSort. It is the best of the known sorting algorithms, guaranteeing $O(n\log n)$ in the worst case, just as *mergeSort*. But it needs in essence no more memory space than the array needs itself (remember that *mergeSort* needs an additional temporary array). The basic idea og *heapSort* is very simple:

1. The n elements to sort are inserted in a heap; this results in complexity $O(n\log n)$.
2. The minimum is deleted n times; complexity $O(n\log n)$.

Let $a = a_0 \dots a_{n-1}$ denote an array of n objects a_i that are to be sorted with respect to their keys a_i.

Definition 4.3. A subarray $a_i \dots a_k$, $1 \leqq i \leqq k \leqq n$, is called a *subheap*, if

$$\left.\begin{array}{ll} a_j \leqq a_{2j} & \text{if } 2j \leqq k, \\ a_j \leqq a_{2j+1} & \text{if } 2j+1 \leqq k. \end{array}\right\} \quad \forall j \in \{i,\dots,k\}. \tag{4.7}$$

□

So if $a = a_0 \dots a_n$ is a subheap, then a is also a heap [8, §4.5.1]. Before we define heapSort, we first introduce two fundamental heap algorithms, insert and deleteMax. They give a feeling for the convenient properties of heaps. Then we consider the algorithm reheap(l,r) which is part of heapSort.

insert and deleteMax

Recall again the essential properties of heaps given in [8, §4.5.1]. Let be h an array of n elements, and let h_i denote its i-th entry. To insert an element we can take the following algorithm:

```
algorithm insert (e) // inserts object e into the heap
n ← n+1;                    // extend the heap with one object
h_n ← e;                    // insert object e
i ← n;                      // start i from the bottom
while (i > 0)               // while root is not reached
    if (h_i > h_⌊(i−1)/2⌋)  // heap property violated?
        h_i ↔ h_⌊(i−1)/2⌋;
        i ← ⌊(i−1)/2⌋;      // go up one generation
    else
        i ← 0;              // exit loop!
```

The effect of this algorithm is shown in figure 4.2.

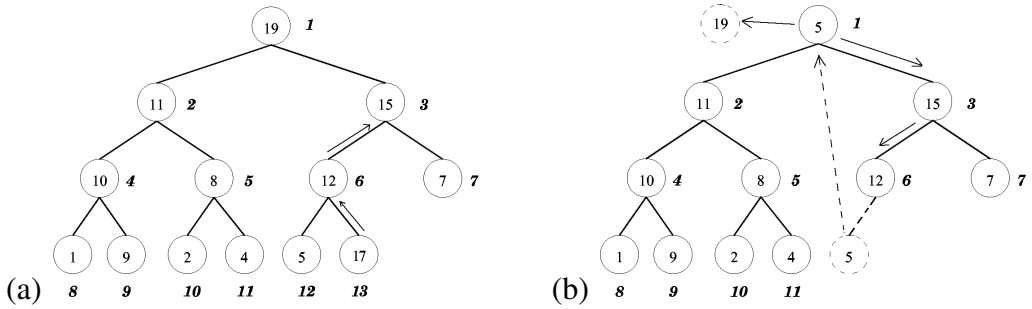


Figure 4.2: (a) The subroutine insert. (b) The subroutine deleteMax.

```
algorithm deleteMax ()      //: deletes the maximum object of the heap H
h_0 ← h_n;                  // maximum now deleted
n ← n−1;                    // decrease heap
i ← 0;                      // start i from the root
while (2i+1 ≦ n)            // while there is at least a left child
    l ← 2i+1;  r ← 2(i+1);  // index of left and right child
    if (r ≦ n)              // does right child exist at all?
        if (h_l > h_r)      // which child is greater?
            max ← l;
        else
            max ← r;
    else
        max ← l;
    if (h_i < h_max)        // heap property violated?
        h_i ↔ h_max;  i ← max;
    else
        i ← n+1;            // exit loop
```

reheap

Algorithm *reheap* lets the element a_l "sink down into" the heap such that a subheap $a_{l+1}, a_{i+1} \dots a_r$ is made to a subheap $a_l, a_{i+1} \dots a_r$.

```
algorithm reheap (l, r)
i ← l;
while (2i+1 ≦ r)
    if (2i+1 < r)
        if (a_{2i+1} > a_{2(i+1)}) // choose index c of greatest child — 1st comparison
            c ← 2i+1;
        else
            c ← 2(i+1);
    else                          // 2i+1 = r, only one child!
        c ← 2i+1;
    if (a_i < a_c)                // necessary to exchange with child? — 2nd comparison
        a_i ↔ a_c;  i ← c;
    else
        i ← r;                    // exit loop!
```

Note by Equation (4.7) in [8] that the left child of node a_i in a heap — if it exists — is a_{2i+1}, and the right child is $a_{2(i+1)}$. Figure 4.3 shows how *reheap* works. Algorithm reheap needs two key

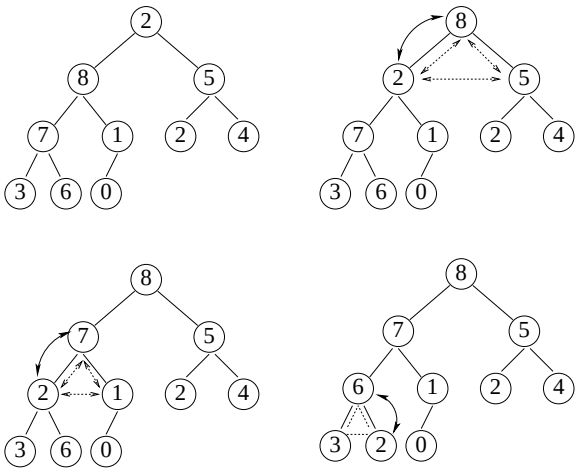


Figure 4.3: The subroutine reheap

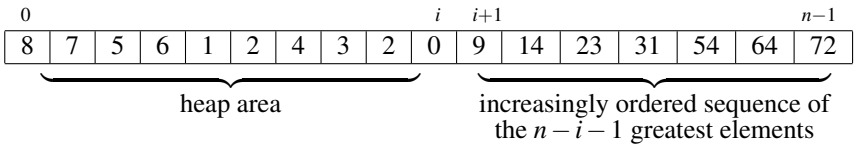
comparisons on each tree level, so at most $2\log n$ comparisons for the whole tree. Therefore, the complexity T_{reheap} of reheap is

$$T_{\text{reheap}}(n) = O(\log n). \tag{4.8}$$

Now we are ready to define algorithm *heapSort* for an array a with n elements, $a = a_0, a_1, \dots, a_{n-1}$. Initially, a need *not* be a heap.

```
algorithm heapSort()
for (i = ⌊(n−1)/2⌋; i ≧ 0; i--)    // phase 1: Building the heap
    reheap(i, n−1);
for (i = n−1; i ≧ 1; i--)          // phase 2: Selecting the maximum
    a_0 ↔ a_i; reheap(0, i−1);
```

How does it work? In phase 1 (building the heap) the subheap $a_{\lfloor (n-1)/2 \rfloor + 1}, \ldots, a_{n-1}$ is extended to the subheap $a_{\lfloor (n-1)/2 \rfloor}, \ldots, a_{n-1}$. The loop is run through $(n/2)$ times, each with effort $O(\log n)$. In phase 2 the sorted sequence is built from the tail part of the array. For this purpose the maximum a_0 is exchanged with a_i, and thus the heap area is reduced by one node to $a_0, \ldots, a_{i-1}$. Because $a_1, \ldots, a_{i-1}$ still is a subheap, reheaping of a_0 makes $a_0, \ldots, a_{i-1}$ a heap again:



In phase 2 the loop will be run through for $(n-1)$ times. Therefore, in total *heapSort* has complexity $O(n \log n)$ in the worst case.

4.5 Comparison of sort algorithms

To summarize, we have the complexities of the various sorting algorithms listed in Table 4.1.

Complexity	selection/insertion/bubble	quick sort	merge sort	heap sort	pigeonhole sort
worst case	$O(n^2)$	$O(n^2)$	$O(n \ln n)$	$O(n \ln n)$	$O(n)$
average case	$O(n^2)$	$O(n \ln n)$	$O(n \ln n)$	$O(n \ln n)$	$O(n)$
space	$O(1)$	$O(\ln n)$	$O(n)$	$O(1)$	$O(n)$

Table 4.1: Complexity and required additional memory space of several sorting algorithms on data structures with n entries; pigeonhole sort is assumed to be applied to integer arrays with positive entries $\leqq O(n)$.

Chapter 5

Searching with a hash table

Is it possible to optimize searching in unsorted data structures? In [8, Satz 4.3] we learned the theoretic result that searching a key in an unsorted data structure is linear in the worst case, i.e., $\Theta(n)$. In Theorem A.5 on p. 110) it is shown that a naive "brute force" search, or: *exhaustion*, costs running time of order $\Theta(n)$ also *on average*. So, these are the mathematical lower bounds which restrict a search and cannot be decreased.

However, there is a subtle backdoor through which at least the average bound can be lowered considerably to a constant, i.e., $O(1)$, albeit to the price of additional calculations. This backdoor is called *hashing*.

The basic idea of hashing is to *calculate* the key from the object to store and to minimize the possible range these keys can attain. The calculation is performed by a hash function. Sloppily said, the hashing principle consists in storing the object chaotically somewhere, but remembering the position by storing the reference in the hash table with the calculated key. Searching the original object one then has to calculate the key value, look it up in the hash table, and get the reference to the object.

The hashing principle is used, for instance, by the Java Collection classes `HashSet` and `HashMap`. The underlying concept of the hash function, however, is used also in totally different areas of computer science such as cryptology. Two examples of hash functions used in cryptology are MD5 and SHA-1. You can find a short introduction to hash functions in German in [20].

5.1 Hash values

5.1.1 Words and alphabets

To write texts we need symbols from an alphabet. These symbols are letters, and they form words. We are going to formally define these notions now.

Definition 5.1. An *alphabet* is a finite nonempty set $\Sigma = \{a_1, \ldots, a_s\}$ with a linear ordering

$$a_1 < a_2 < \cdots < a_s.$$

Its elements a_i are called *letters* (also *symbols*, or *signs*). □

Example 5.2. (i) A well-known alphabet is $\Sigma = \{\text{A, B, C, } \ldots, \text{Z}\}$. It has 26 letters.
(ii) In computing the binary alphabet $\Sigma = \{0, 1\}$ is used. It has two letters. □

Definition 5.3. Let $\Sigma = \{a_1, \ldots, a_s\}$ be an alphabet.
(i) A *word* (or a *string*) over Σ is a finite sequence of letters, such as

$$w = a_{i_1} a_{i_2} \ldots a_{i_n} \qquad (i_j \in \{1, \ldots, s\}.$$

The *length n* of a word is the number of its (nonempty) letters.

(ii) The *empty word* is defined as λ.

(iii) The set of words of words of length $n \in \mathbb{N}$ is denoted by Σ^n. The set of all possible words over Σ, including the empty word λ, is denoted Σ^*. It is often called the *universe*. □

Notice that $\Sigma^n \subset \Sigma^*$ for any $n \in \mathbb{N}$.

Example 5.4. (i) A word over the alphabet Σ of example 5.2 (i) is, e.g., NOVEMBER. It has length 8, i.e.

$$\text{NOVEMBER} \in \Sigma^8.$$

(ii) A word over the binary alphabet $\Sigma = \{0,1\}$ is $1001001 \in \{0,1\}^7$. □

Because alphabets are finite sets, their letters can be identified with natural numbers. If an alphabet has m letters, its letters can be identified ("coded") with the numbers

$$\mathbb{Z}_m = \{0,1,\ldots,m-1\}. \tag{5.1}$$

For instance, for the 26-letter alphabet Σ of example 5.2 (i) we can choose the code $\langle\cdot\rangle : \Sigma \to \mathbb{Z}_{26}$, given by

a_i	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\langle a_i \rangle$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

(5.2)

That means, $\langle \text{N} \rangle = 14$. Another example is the 127-letter alphabet of the ASCII-Code, where e.g. $\langle \text{A} \rangle = 65$, $\langle \text{N} \rangle = 78$, or $\langle \text{a} \rangle = 97$. A generalization is *Unicode* which codifies $2^{16} = 65\,536$ letters. For notational convenience the 2^{16} numbers are usually written in their hexadecimal representation with four digits (note: $2^{16} = 16^4$), i.e.

$$\mathbb{Z}_{(2^{16})} = \{0000_{16}, 0001_{16}, 0002_{16}, \ldots, \text{FFFF}_{16}\} \tag{5.3}$$

The first 256 letters and their hexadecimal code is given in figure 5.1.

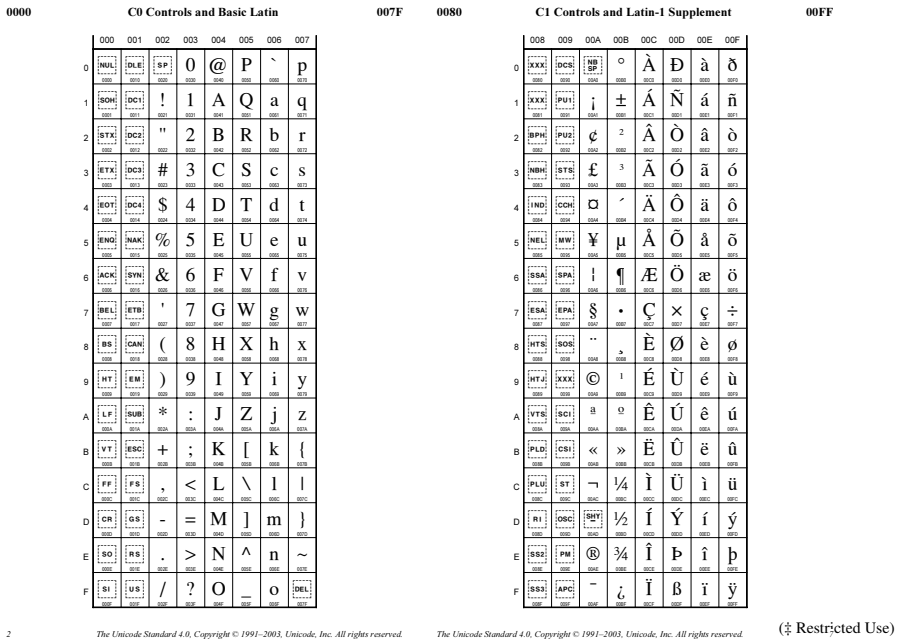


Figure 5.1: The first code numbers of Unicode.

5.1.2 Hash functions

Definition 5.5. A *hash function* is a function $h : W \to H$ of a (possibly infinite) set $W \subset \Sigma^*$ of words onto a finite set $H \subset \mathbb{Z}$ of integers called *hash values*, such that the following properties hold:

- $h(w)$ is easily computable, i.e., computable by an efficient ("very fast") algorithm;
- to a given hash value y, it is hard to find a word w such that $h(w) = y$;
- to a given word w, it is hard to find a second word w' such that $h(w) = h(w')$, i.e., a second word with the same hash value.

If two different words have the same hash value, we have a *collision*. The set of all possible hash values is also called the *hash table*, and the number of all hash values its *capacity*. Sometimes, for instance in the Java-API, the hash values are also called *buckets*. □

Example 5.6. Let $h : \{0,1\}^* \to \{0,1\}$,

$$h(w) = w_n \oplus \ldots \oplus w_1$$

be the XOR operation of an arbitrarily long bit string. For instance, $h(101) = 1 \oplus 0 \oplus 1 = 0$. Then h is a (very simple) hash function, and 0 is the hash value of 101. The input length is arbitrary, but the output is either 0 or 1, i.e., 1 bit. Since $h(1001) = 0$, the two different words $w^{(1)} = 101$ and $w^{(2)} = 1001$ have the same hash value. Thus we have a collision. □

Example 5.7. The last digit of the 13-digit ISBN[1] is a hash value computed from the first 12 digits and is called "check digit." To date, the first three digits are 978 or 979, and may be different according to the EAN system,

$$978 w_4 w_5 \ldots w_{12} h.$$

Let $\Sigma = \{0,1,\ldots,9\}$. Then the first 12 digits of the ISBN form a word $w \in \Sigma^{12}$, and the last digit is given as $h(w)$ where the hash function $h : \Sigma^{12} \to \Sigma$

$$h(w_1 w_2 \ldots w_{12}) = -\sum_{i=1}^{12} g_i \cdot w_i \bmod 10 \qquad \text{where } g_i = 2 + (-1)^i = \begin{cases} 1 & \text{if } i \text{ is odd,} \\ 3 & \text{if } i \text{ is even.} \end{cases}$$

For example,

$$h(978389821656) = -138 \bmod 10 = 2,$$

since

9	7	8	3	8	9	8	2	1	6	5	6	
1	3	1	3	1	3	1	3	1	3	1	3	
9	21	8	9	8	27	8	6	1	18	5	18	$\sum 138.$

Therefore 978-3-89821-656-2 is a valid ISBN number. □

A hash function cannot be invertible, since it maps a huge set of words onto a relatively small set of hash values. Thus there *must* be several words with the same hash value, forming a collision.

Hash functions are used in quite different areas. They do not only play an important role in the theory of data bases, but are also essential for digital signatures in cryptology. For instance, they provide an invaluable technique to support reliable communications. Consider a given

[1] also called ISBN-13, valid since 1 January 2007; *http://www.isbn-international.org/*

message m which shall be transmitted over a channel; think, for instance, of IP data packets sent through the internet or a bit string transmitted in a data bus in your computer. Most channles are noisy and may modify or damage the original message. In the worst case the receiver does not notice that the data are corrupted and relies on wrong information.

A quick way to enable the receiver to check the incoming data is to send along with the message w its hash value $h(w)$, i.e., to send

$$(w, h(w)).$$

If sender and receiver agree upon the hash function, then the receiver can check the data consistency by simply taken the received message m', compute its hash value $h(w')$, and compare it to the received hash value $h(w)$. If the transmission has been modified during the transmission, and the hash function is "good" enough, then the receiver notices a difference and may contact the sender to resend him the message.

This is realized very often in communication channels. In case of IP packets or in the data bus of your computer, the hash function is a simple bitwise parity check, in cryptographic communications it is a much more complex function such as SHA-1. A short survey of important hash functions used in cryptology is given in Table 5.1. Notably, each of them base on MD4

Hash Function	**Block Length**	**Relative Running Time**
MD4	128 bit	1,00
MD5	128 bit	0,68
RIPEMD-128	128 bit	0,39
SHA-1	160 bit	0,28
RIPEMD-160	160 bit	0,24

Table 5.1: Standard hash functions, according to [4].

which has been developed by Ron Rivest at the end of the 1980's. RIPEMD-160 is supposed to be very secure. SHA-1 is the current international standard hash function in cryptography.

Example 5.8. SHA *(Secure Hash Algorithm)* has been developed by the NIST and the NSA and is the current standard hash function. It works on the set Σ^* of arbitrary words over the binary alphabet $\Sigma = \{0,1\}$ and computes hash values of fixed length $m = 160$ bit in binary format with leading zeros, i.e.,

$$\mathrm{SHA} : \{0,1\}^* \to \{0,1\}^{160}. \tag{5.4}$$

For a given binary word $w \in \{0,1\}^*$ it performs the following steps.

1. *Divide the bit word w into blocks of 512 bit:* The word w is padded such that its length is a multiple of 512 bits. More precisely, the binary word is attached by a 1 and as many 0's such that its length is a multiple of 512 bits minus 64 bits, added by a 64-bit-representation of the (original) word.

2. *Form 80 words à 32 bits:* Each 512-bit block is divided into 16 blocks $M_0, M_1, \ldots, M_{15}$ à 32 bit which are transformed into 80 words $W_0, \ldots, W_{79}$ according to

$$W_t = \begin{cases} M_t & \text{if } 0 \leqq t \leqq 15, \\ (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1 & \text{otherwise.} \end{cases}$$

   Here $\lll$ denotes the *bit rotation*, or *circular left-shifting* (e.g., $\texttt{10100} \lll \texttt{1} = \texttt{01001}$).[2]

[2]The original specification of SHA as published by the NSA did not contain the bit rotation. It corrected a "technical problem" by which the standard was less secure than originally intended [37, S. 506]. To my knowledge, the NSA never has explained the nature of the problem in any detail.

3. *Initialize the variables and constants:* In SHA there are used 80 constants $K_0, \ldots, K_{79}$ (with only four different values), given by

$$K_t = \begin{cases} \texttt{0x5A827999} = \lfloor \sqrt{2} \cdot 2^{30} \rfloor & \text{wenn } 0 \leqq t \leqq 19, \\ \texttt{0x6ED9EBA1} = \lfloor \sqrt{3} \cdot 2^{30} \rfloor & \text{wenn } 20 \leqq t \leqq 39, \\ \texttt{0x8F1BBCDC} = \lfloor \sqrt{5} \cdot 2^{30} \rfloor & \text{wenn } 40 \leqq t \leqq 59, \\ \texttt{0xCA62C1D6} = \lfloor \sqrt{10} \cdot 2^{30} \rfloor & \text{wenn } 60 \leqq t \leqq 79, \end{cases}$$

five constants $A, \ldots, B$ given by

$$A = \texttt{0x67452301}, \qquad B = \texttt{0xEFCDAB89}, \qquad C = \texttt{0x98BADCFE},$$

$$D = \texttt{0x10325476}, \qquad E = \texttt{0xC3D2E1F0}.$$

and five variables[3] $a, \ldots, e$, being initialized as

$$a = A, \qquad b = B, \qquad c = C, \qquad d = D, \qquad e = E.$$

Each constant and each variable has 32 bit = 8 Bytes.

4. *The main loop:*

for ($t = 0; t \leqq 79; t$++) {
  tmp $= (a \lll 5) + f_t(b,c,d) + e + W_t + K_t$;
  $e = d$;
  $d = c$;
  $c = b \lll 30$;
  $b = a$;
  $a =$ tmp;
}

Here the family of nonlinear functions f_t is defined by

$$f_t(x,y,z) = \begin{cases} (x \wedge y) \vee (\neg x \wedge z) & \text{if } 0 \leqq t \leqq 19, \\ (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) & \text{if } 40 \leqq t \leqq 59, \\ x \oplus y \oplus z & \text{otherwise.} \end{cases}$$

□

5.2 The hashing principle

The basic idea of *hashing* is quite different. First, the dictionary is implemented as an *unsorted* array of size n. The address of each word in the dictionary is stored in a smaller array t of m pointers with $m \leqq n$, the so-called *hash table*. Second, the address of each word w is *calculated* by a function

$$h : U \to \{0, \ldots, m-1\}, \quad w \mapsto h(w)$$

which assigns to each word w a certain index $h(w)$ in the hash table. h is called *hash function*, and $h(w)$ is called *hash value*. The principle is illustrated in figure 5.2. To use a picture, the hash function distributes the N words into m containers. Each container is an entry of the hash table.

[3] MD5 consists of nearly the same instructions as SHA, but has only 64 constants $K_i = \lfloor 2^{32} |\sin i| \rfloor$, only four constants $A, \ldots, D$ and four variables (actually, it computes a hash value of only 128 bits!).

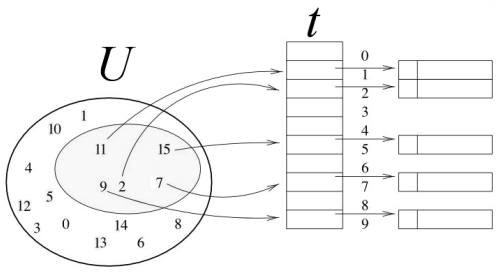


Figure 5.2: Hashing principle. Here the universe $U = \mathbb{Z}_{16} = \{0, 1, \ldots, 15\}$, the hash table t with $m = 10$ entries, and the hash function $h(w) = w \bmod 10$.

5.3 Collisions

We have a principal problem with hash functions. The domain of definition is a huge sets of words of size N, whereas the number of address items m usually is much smaller, $m \ll N$. That means it may come to the effect that various different words obtain the same hash value. As we defined above, such an event is called a *collision*. Let us examine the following example.

Example 5.9. We now construct a simple hash table. Let be the universe be

$$U = \{22, 29, 33, 47, 53, 59, 67, 72, 84, 91\}.$$

Moreover let $h : U \to \mathbb{Z}_{11}$ be the hash function $h(w) = w \bmod 11$. Then we calculate the hash table

$h(w)$	w
0	33, 22
1	67
2	
3	91, 47
4	59
5	
6	72
7	29, 84
8	
9	
10	

□

The example demonstrates that relatively many collisions can occur even though $m = N$, i.e. even though there are as many addresses as words! This at first glance surprising fact is closely related to the famous "birthday paradox" we explain later on.

How probable are collisions? We assume an "ideal" hash function distributing the n words equally probable on the m hash values. Let be $n \leqq m$ (because for $n > m$ a collision *must* occur!). Denote

$$p(m,n) = \text{probability for at least one collision in } n \text{ words and } m \text{ hash values.}$$

(In the sequel we will often shortly write "p" instead of "$p(m,n)$.") Then the probability q that *no* collision occurs is

$$q = 1 - p. \tag{5.5}$$

We first will calculate q, and then deduce p from q. So, what is q? If we denote by q_i the probability that the i-th word is mapped to a hash value without a collision under the condition that all the former words are valued collisionless, then

$$q = q_1 \cdot q_2 \cdot \ldots \cdot q_n.$$

First we see that $q_1 = 1$, because initially all hash values are vacant and the first word can be mapped on any value without collision. However, the second word finds one hash value occupied and $m-1$ vacant values. Therefore $q_2 = (m-1)/m$. So we found generally that

$$q_i = \frac{m-i+1}{m} \qquad 1 \leqq i \leqq n,$$

because the i-th word finds $(i-1)$ values occupied. Thus we have for p

$$\boxed{p = 1 - \frac{m(m-1)(m-2)\cdots(m-n+1)}{m^n}} \tag{5.6}$$

In table 5.2 there are numerical examples for $m = 365$. It shows that only 23 words have to be present such that a collision occurs with a probability $p > 0.5$! For 50 words, the probability is 97%, i.e. a collision occurs almost unavoidably. The Hungarian-American mathematician

n	$p(365, n)$
22	0.476
23	0.507
50	0.970

m	$1.18\sqrt{m}$
365	22.49
1 000 000	1177.41
$2^{128} \approx 3 \cdot 10^{38}$	$2.2 \cdot 10^{19}$

Table 5.2: The probability p for collision occurence for $m = 365$ (left) and Halmos estimates for some hash capacities m

Paul Halmos (1916–2006, *"Computers are important — but not for mathematics,"* [22, p. 31]) computed the estimate $n \approx 1.18\sqrt{m}$ for the number n of words such that $p(m,n) > 1/2$ [22, pp. 31].

Example 5.10. Birthday paradox. Suppose a group of n people in a room. How probable is it that at least two people have the same birthday? In fact, this question is equivalent to the collision problem above. Here the number n of words corresponds to the number of persons, and the possible birthdays corresponds to $m = 365$ hash values. Thus the table 5.2 also gives an answer to the birthday paradox: For 23 persons in a room the probability that two have the same birthday is greater than 50%! □

5.3.1 Strategies of collision resolution

Once we have put up with the fact that hashing collisions occur with high probability — even for comparably small numbers of words to insert into a dictionary —, we have to think about strategies to handle with collisions.

Hashing with chaining

A solid method to resolve collisions is to create a linked list at each hash table entry. Then any new word w will be appended to the list of its hash value $h(w)$. This is the *collision resolution by chaining*. To continue with example 5.9 above, we obtain the hash table in figure 5.3.

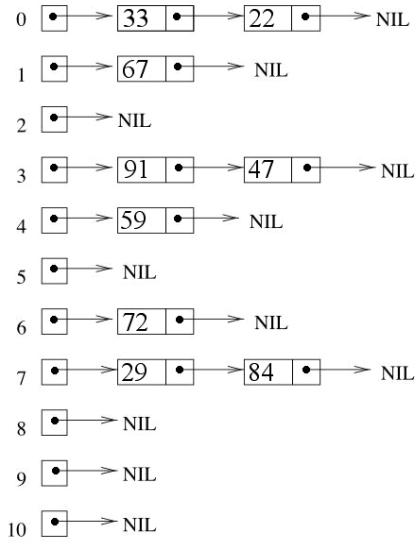


Figure 5.3: Hashing with linked lists.

Complexity analysis. Assume we want to insert n words into a hash table of size m. For all three operations *insert*, *delete*, and *member* of word w the linked list at the hash entry $t[h(w)]$ must be run through. In the worst case all words obtain the same hash value. Searching words in the hash table then has the same time complexity as running through a linked list with n objects, i.e.

$$T_{\text{worst}}(n) = O(n).$$

However, we will see that hash table have a average case complexity. To start the analysis, we first consider the question: How much time does an search take? The average length of a linked list is n/m. The running time of computing the hash function is a constant, i.e. $O(1)$. Adding both running times for an average case search yields the complexity $T_{\text{mean}}(n) = O(1+n/m)$.

Theorem 5.11. *The average complexity of the three dictionary algorithms* insert, delete, *and* member *of a hashing with linked lists (separate chaining) is*

$$T_{\text{mean}}(n) = O(1+\alpha) \tag{5.7}$$

Here α denotes the load factor *given by*

$$\alpha = \frac{n}{m} \tag{5.8}$$

where m denotes the size of the hash table, and n the inserted words. The worst case complexity is

$$T_{\text{worst}}(n) = O(n). \tag{5.9}$$

Hashing with open addressing

A second strategy to resolve collisions is to find a free hash value, i.e. a vacant place in the hash table. A great advantage with respect to hashing with chaining is the avoidance of linked lists, leaving us in hope of a better worst case running time. But there are two main disadvantages. First the load factor α must not exceed 1 (if $\alpha > 1$, the n words to be inserted are more than m, i.e. the hash table is full). Second, the deletion of must be done carefully.

As an example, consider the universe $U = \{39, 43, 61, 67, 75\}$ and the hash table with 11 hash values $\{0, 1, \ldots, 10\}$ and the hash function is $h(w) = w \bmod 11$. A collision occurs for

$v = 61$ and $w = 39$, since $h(v) = h(w) = 6$. There are mainly two ways to calculate a new hash value in case a collision is detected.

1. *(Double hashing)* Use two hash functions $h(w), h'(w)$ mod m and try the hash values

$$h_i(w) = h(w) + ih'(w)$$

for $i = 0, 1, 2, \ldots, m-1$ one after the other, until a free value is found.

2. Use hash m functions $h_i(w)$, $i = 0, 1, \ldots, m-1$, and try the hash values

$$h_0(w),\ h_1(w),\ h_2(w),\ \ldots,\ h_{m-1}(w)$$

until a free one is found.

There is one great problem for hashing with open addressing, concerning the deletion of words. If a word w is simply deleted, a word v that has been passed over w because of a collision could not be found! Instead, the cell where w is located has to be marked as deleted but cannot be simply released for a new word.

Therefore, hashing with open addressing is not appropriate for

- very "dynamical" applications where there are lots of inserts and deletes;
- for cases in which the number n of words to be inserted is greater than the hash table.

Complexity analysis. We assume that the hash function sequence h_i is "ideal" in the sense that the sequence $h_0(w), h_1(w), \ldots, h_{m-1}(w)$ is uniformly distributed over the possible hash values. In this case we speak of *uniform hashing.* Then we have the following theoretical result.

Theorem 5.12. *Let be h_i an ideal hash function sequence for m hash values, where already n values are already occupied. Then the expected costs (numbers of probes) is approximately*

$$C'_n = \frac{1}{1-\alpha} \qquad \textit{for insert and unsuccessful search} \tag{5.10}$$

$$C_n = \frac{1}{\alpha} \cdot \ln \frac{1}{\alpha}. \qquad \textit{for insert and successful search} \tag{5.11}$$

Here again $\alpha = n/m$ is the load factor.

For the proof see [19] pp.101.

collision resolution strategy	complexity
chaining	$O(1+\alpha)$
open addressing (unseccessful search)	$O(\frac{1}{1-\alpha})$
open addressing (successful search)	$O(\frac{1}{\alpha} \cdot \ln \frac{1}{\alpha})$

Table 5.3: Complexities of collision resolution strategies

Hash functions for collision resolution. Let the number m of possible hash values be given.

- *Linear probing.* One common method is the *linear probing*. Suppose a hash function $h(w)$.
$$h_i(w) = \Big(h(w) + i\Big) \bmod m, \qquad i = 0, 1, \ldots, m-1. \tag{5.12}$$

- *Quadratic probing.* Suppose a hash function $h(w)$.
$$h_i(w) = \Big(h(w) + i^2\Big) \bmod m. \tag{5.13}$$

- *Double Hashing.* Suppose two hash functions $h(w), h'(w)$. Then we define a sequence of hash functions
$$h_i(w) = \Big(h(w) + h'(w) \cdot i^2\Big) \bmod m, \qquad i = 0, 1, \ldots, m-1. \tag{5.14}$$
We require that the two hash functions are *(stochastically) independent* and *uniform*. This means that for two different words $v \neq w$ the events
$$X = \text{“}h(v) = h(w)\text{”} \qquad \text{and} \qquad X' = \text{“}h'(v) = h'(w)\text{”}$$
each occur with probability $1/m$, and both events together occur with probability $1/m^2$; or expressed in formulae:
$$P(X) = \frac{1}{m}, \qquad P(X') = \frac{1}{m}, \qquad P(X \wedge X') = P(X) \cdot P(X') = \frac{1}{m^2}.$$
This yields a real excellent hash function! Experiments show that this function has running times that are practically not distinguishable from ideal hashing. However, it is not easy to find appropriate pairs of hash functions which can be *proved* to be independent. Some are given in [29] pp.528

Part II

Optimization and Networks

Chapter 6

Optimization problems

In this chapter we present the definition and formal structure of optimization problems and give some typical examples.

6.1 Examples

Example 6.1. (Regression polynomial) In statistics, one is often interested in finding a *regression polynomial*, or *regression curve*, for a given series of data pairs $(t_1, y_1), \ldots, (t_N, y_N)$ where[1] $t_i, y_i \in \mathbb{R}$ for $i = 1, \ldots, N$. Such data may represent measurement samples with uncertainties due to the measurement apparatus or signal perturbations by noise. To find a regression polynomial of degree $n-1$ to these data we mean that we want to specify the p real coefficients $x_0, x_1, \ldots, x_{n-1}$ of a polynomial $p : \mathbb{R} \to \mathbb{R}$,

$$p(t) = x_0 + x_1 t + x_2 t^2 + \cdots + x_{n-1} t^{n-1} \tag{6.1}$$

such that $y_i \approx p(t_i)$ for all $i = 1, \ldots, N$. For instance, if we want to find a linear regression polynomial, we have $p = 2$ and look for two parameters x_0, x_1 such that $y_i \approx x_0 + x_1 t_i$ (Figure 6.1).

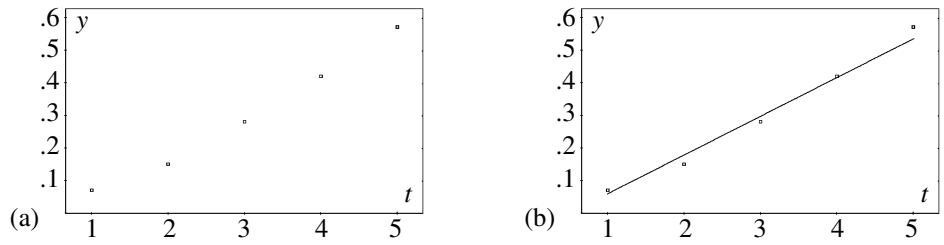


Figure 6.1: (a) Scatterplot of the data sample of the sample $(1, .07)$, $(2, .15)$, $(3, .28)$, $(4, .42)$, $(5, .57)$. (b) Linear regression of the sample.

A data pair series with $t_i = i$ especially represents a *time series*, i.e., a series of data values $(y_1, y_2, \ldots, y_N)$, where y_t denotes the data value measured at time or period t. For instance, these data may represent sales figures of a certain article at different periods. Specifying a regression polynomial offers the possibility to gain from past sales figures a forecast for the next few periods. The linear regression line, e.g., represents the bias, or "trend line" [40]. □

Example 6.2. (Traveling salesman problem (TSP)) A traveling salesman must visit n cities such that the round trip is as short as possible. Here "short" may be meant with respect to time or with respect to distance, depending on the instance of the problem. The TSP is one of the most important — and by the way one of the hardest — optimization problems. It has many

[1] The problem could easily be generalized to the case $t_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}^k$ with $m, k \in \mathbb{N}$.

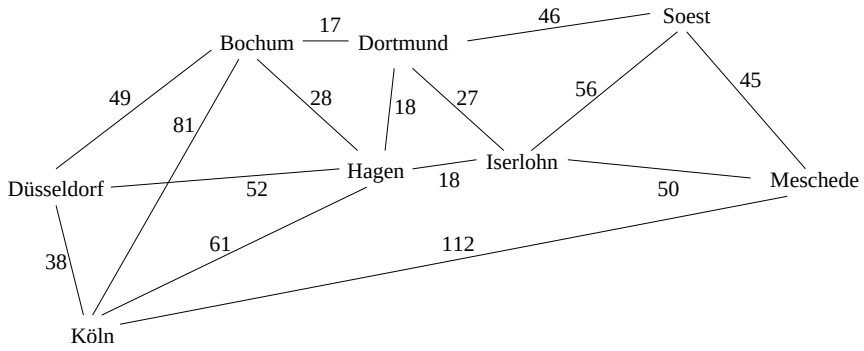


Figure 6.2: A TSP for $n = 8$ cities. What is the shortest round-trip for the traveling salesman visiting each city exactly once, starting and terminating in Hagen?

applications. For instance, a transport service which has to deliver goods at different places may be considered as a TSP; another example of a TSP is the problem to program a robot to drill thousands of holes into a circuit board as quick as possible.

There are many generalizations of the TSP, for example the travel time between two cities may depend on time, such as the rush hours where it is longer than at night. □

Example 6.3. (Production planning) A company produces n products $P_1, \ldots, P_n$, gaining a specific profit for each product. To manufacture these products, the company has available m_1 machines each of which has a limited total production time and specific manufacturing times for each product, as well as m_2 resource materials which are required in specific portions for each product but which are only disposable in limited certain quantities. How much, then, of each product, measured in quantity units per period (e.g., kg/day), should be manufactured to maximize the profit? □

6.2 The general structure of optimization problems

Before we consider strategies to solve optimization problems, we first study the general structure of such problems. What do all optimization problem do have in common? To answer this question, we have to formalize a general optimization problem and point out its essential elements. In mathematics, the term *optimization* refers to the finding of an optimum for a real-valued function $f : S \to \mathbb{R}$ on a given domain S of possible solutions. Usually, such a searched optimum is a global maximum or minimum, and the domain underlies some given constraints. In symbols, an optimization problem is given by

$$\boxed{f : S \to \mathbb{R}, \qquad f(x) \longrightarrow \max \quad \text{for } x \in S} \tag{6.2}$$

for a *maximum problem*, for example optimizing gain, and

$$\boxed{f : S \to \mathbb{R}, \qquad f(x) \longrightarrow \min \quad \text{for } x \in S} \tag{6.3}$$

for a *minimum problem*, for example optimizing costs. The domain S is called the *search space* of the optimization problem, and the function f is the *objective function*, or *cost function*. In the next paragraphs, we will consider these notions in more detail.

6.2.1 The search space

The first thing to formulate an optimization problem is to specify the *search space* S. It is the set of all feasible solutions, or *candidate solutions*. Typically, S is some subset of $\mathbb{R}^n$ (the

n-dimensional *Euclidean space* or *hyperspace*), or of $\mathbb{Z}^n$, where n denotes the number of parameters which have to be adjusted to yield the optimum. That is, a feasible solution $x \in S$ is then given as

$$x = (x_1, x_2, \ldots, x_n).$$

Therefore, x is an n-dimensional vector if $S \subset \mathbb{R}^n$. On the other hand, if $S \subset \mathbb{Z}^n$, the parameters of a feasible solution x can take only integer values and thus x is an n-dimensional lattice point ("a vector with integer coefficients"). Moreover, the optimization problem is called *discrete* or *combinatorial* in this case.

For some optimization problems, the search space is the total Euclidean space $\mathbb{R}^n$ or the total integer lattice $\mathbb{Z}^n$. More often, however, the possible solutions are restricted by certain conditions, the *constraints*. Such constraint may be more or less complicated, but their consequence is always that S is a proper subset of $\mathbb{R}^n$ or $\mathbb{Z}^n$, respectively. Moreover, in almost all cases a discrete optimization problem has a search space with finitely many feasible solutions.

The determination of the search space for a given problem is the first step in its mathematical formulation. Of course, a given problem may be represented by several search spaces. Sometimes a search space might be easily or even naturally be derived from the given problem at hand, but sometimes it may be a really hard task to find an appropriate search space. In fact, the chosen search space is crucial for the implementation of an optimization problem on a computer, because it specifies the data structure which is used by an algorithm to execute the optimization.

Example 6.1 (Regression, continued). To determine the regression polynomial of a data pair series (t_1, y_1), ..., (t_N, y_N), the degree of the polynomial we wish to obtain is essential for the number of parameters to adjust: If we want to determine a regression polynomial of degree $(p-1)$ according to Equation (6.1), we need n parameters, since we search the p coefficients x_0, x_1, ..., x_{n-1} of the polynomial $p(t)$. Therefore, the search space for the regression polynomial of degree $(n-1)$ is

$$S = \mathbb{R}^n. \tag{6.4}$$

For instance, the search space of a linear regression is the real plane, $S = \mathbb{R}^2$. □

Example 6.2 (TSP, continued). If we number the n cities which the traveling salesman has to visit by 1, 2, ..., n, then the search space of the TSP is given by the set

$$S = \{(x_1, \ldots, x_n) : x_i \in \{1, 2, \ldots, n\}, x_i \neq x_j \text{ for } i \neq j\} \tag{6.5}$$

Here the vector $x = (x_1, x_2, \ldots, x_n)$ of integers represents the round trip starting in city x_1, then proceeding to city x_2, and so on, until reaching city x_n as the last city before returning to the start x_1. The condition $x_i \neq x_j$ for $i \neq j$ guarantees that no city is visited twice during the trip. In fact, the search space of the TSP is the set of all "permutations" of $\{1, \ldots, n\}$. Since furthermore $S \subset \mathbb{Z}^n$, the TSP is a discrete optimization problem. □

Example 6.3 (Production planning, continued). What is searched are the n individual quantities of the products. Let denote x_i be the quantity per period which is produced of product P_i, and $x = (x_1, \ldots, x_n)$. Then the capacities of the m_1 machines and the restrictions of the m_2 resources impose $m = m_1 + m_2$ constraints $f_j(x) \leqq f_j^{\max}$, $j = 1, \ldots, m$, as limits on x. Therefore, the search space of this problem is

$$S = \{x \in \mathbb{R}^n : f_1(x) \leqq f_j^{\max}, \ldots, f_m(x) \leqq f_m^{\max}\} \tag{6.6}$$

Note that we assumed that the quantity values are real values. In principle, we could restrict the optimization problem to integer values, i.e., if we are interested only in the integer numbers of pieces. This then is a combinatorial problem, and we will see below that it is much harder than the continuous problem. □

6.2.2 The objective function

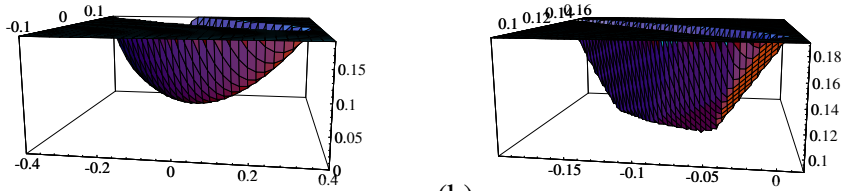
The search space specifies the structure of possible solutions of the optimization problem. However, it does not distinguish an optimum solution. What we need is a function to evaluate each solution with respect to the problem, i.e., a characteristic number by which different solutions are comparable. This evaluation is done by the objective function $f : S \to \mathbb{R}$ which associates to each feasible solution a real number. Depending on whether $f(x)$ evaluates the quality of the feasible solution $x \in S$ or its defect, the optimum is a maximum or a minimum.

The choice of an objective function for a given optimum problem often is not obvious or unique. Different functions may be equally plausible, but yield completely different optima.

Example 6.1 (Regression, continued). A reasonable objective function for the regression problem is the the error function. It sums the distances $p(t_k) - y_k$ of each sample point (t_k, y_k), where p is regression polynomial (6.1) and $k = 1, \ldots, N$. Because a solution is better if and only if the error is smaller, the regression problem is a minimum problem. But what does "distance" exactly mean? A widely used distance measure is the mean squared error of the regression polynomial (6.1) and the sample data,

$$f(x_0, \ldots, x_n) = \frac{1}{N} \sum_{k=1}^{N} [\underbrace{x_0 + x_1 t_k + \cdots + x_{n-1} t_k^{n-1}}_{p(t_k)} - y_k]^2. \tag{6.7}$$

For instance, the error function of the linear regression is given by $f(x_0, x_1) = \frac{1}{N} \sum_{k=1}^{N} [x_0 + x_1 t_k - y_k]^2$ cf. Figure 6.3. It can be solved by calculating the gradient and setting it to zero.



(a) (b)

Figure 6.3: (a) The squared error objective function of linear regression (6.7) for the sample $(1,.07)$, $(2,.15)$, $(3,.28)$, $(4,.42)$, $(5,.57)$; the search space is $S = \mathbb{R}^2$, the minimum is reached at $x = (x_0, x_1) = (-.00068, .1016)$; (b) the absolute-distance error objective function (6.8) for the same sample.

We will not go into more detail of the solution of this problem here, this is done in statistics.[2] Another important error functions is the mean absolute distance

$$f(x) = \frac{1}{N} \sum_{k=1}^{N} |p(t_k) - y_k| \tag{6.8}$$

where $x = (x_0, \ldots, x_{n-1})$. An objective function basing on the mean absolute distance is less influenced by extreme outliers than a objective function using the mean squared error. This is the reason why the mean absolute distance is usually preferred in economical applications, for

[2] To shortly mention at least the simplest case: The linear regression parameters are given by

$$x_0 = \bar{y} - x_1 \bar{t}, \qquad x_1 = \frac{\mathrm{cov}(T, Y)}{\sigma^2(T)}$$

where $\bar{t} = \frac{1}{N} \sum_{k=1}^{N} t_k$ and $\bar{y} = \frac{1}{N} \sum_{k=1}^{N} y_k$ are the mean values, respectively, $\mathrm{cov}(T, Y) = \left(\frac{1}{N} \sum_1^N t_k y_k\right) - \bar{t}\bar{y}$ is the covariance, and $\sigma^2(T) = \frac{1}{N-1} \sum_1^N (t_k - \bar{t})^2$ is the variance [12, §3.1], [42, §2.4].

there often are extreme outliers given as peaks in the sales figures (e.g., because of Christmas trade) or by production downtimes. □

Example 6.2 (TSP, continued). For the TSP, the objective function is quite obvious, it is the total length of a round-trip. Usually, the distances between two directly connected cities are given by a matrix $G = (g_{ij})$ where g_{ij} denotes the distance from city i to city j; we have $g_{ii} = 0$, and $g_{ij} = \infty$ if there is no edge between city i and j. The matrix G is often called the *weight matrix*. Then the objective function of the TSP is $f : S \to \mathbb{R}^+$,

$$f(x) = \sum_{k=1}^{N} g_{x_{k-1}x_k} \tag{6.9}$$

where $x = (x_1, \ldots, x_n)$, and x_1 is the index of the "home town" of the salesman. □

Example 6.3 (Production planning, continued). For the production planning problem, the objective function is naturally given, since it is given by the profit. If product P_k is produced with quantity x_k and yields a specific profit of c_k currency units per quantity unit, then the total profit is given by $f : S \to \mathbb{R}$,

$$f(x) = \sum_{i=1}^{n} c_k x_k. \tag{6.10}$$

□

Example 6.4. (Rastrigin function) The *Rastrigin Function* is an example of a non-linear function with several local minima and maxima. It has been first proposed by Rastrigin as a 2-dimensional function [41]. It reads

$$f(x) = an + \sum_{i=1}^{n} x_i^2 - a\cos(\omega x_i) \tag{6.11}$$

with the external parameters a, $\omega \in \mathbb{R}^+$, and $x = (x_1, x_2, \ldots, x_n)$. The surface of the function

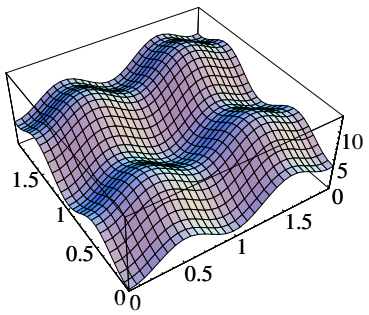


Figure 6.4: The Rastrigin function (6.11) with the search space $S = [0,2]^2 \subset \mathbb{R}^2$ and the external parameter values $a = 2$, $\omega = 2\pi$. The global minimum is at $x = (x_1, x_2) = (0,0)$, its global maximum at $x = (\frac{3}{2}, \frac{3}{2})$.

is determined by the external variables a and ω, which control the amplitude and frequency modulation, respectively. □

Multi-criterion optimization

Many every-day optimization problems are to solve not only with respect to a single criterion but to several criteria. If you want to buy a car, say, you try to optimize some criteria simultaneously, such as a low price, a low mileage, and a high speed.

In general, a multi-criterion optimization consists of m objective functions $f_k : S \to \mathbb{R}$, where $k = 1, \ldots, m$. A usual way to combine these single criteria to a total objective function $f : S \to \mathbb{R}$ by forming a weighted sum,

$$f(x) = \sum_{k=1}^{m} w_k f_k(x) \tag{6.12}$$

with the weights $w_k \in \mathbb{R}$. If the total objective function f is to be maximized, then those single objective functions to be maximized also have a positive weight $w_k > 0$, whereas those criteria to be minimized get a negative weight $w_k < 0$. In the case of a minimizing total objectiv function, the signs are *vice versa*.

We only mention that there is often used another approach to solve multi-criterion problems, namely "Pareto optimality" [15, §6.3.5]. However, we will not consider this subject in the sequel.

6.3 Approaches to solve optimization problems

In general, an optimization problem is mathematically specified by a search space S and a objective function $f : S \to \mathbb{R}$ evaluating each solution. Therefore, the optimization problem is solved by finding optima of the objective function.

Although usually there are more than two parameters to be optimized, Figure 6.4 gives a good intuition about the geometry of the problem. Any objective function represents a "landscape" with mountains and valleys over the search space. In the case of a discrete optimization problem, this is a lattice of integer values. (If we have more than two parameters, we cannot depict the graph of f because our visial system is only adapted to three dimensions; however, mathematically there is nothing particular to the case $n = 2$.)

In this landscape, mountains refer to local maxima, valleys to local minima. Thus a maximum problem is solved if the highest mountain, or one of the highest mountains, is found: its coordinates then give the optimal solution parameters. In case of a minimum problem, it is the lowest valley, or one of the lowest valleys, which is searched for.

There are several approaches to solve a given optimization problem. The most important ones are the following.

6.3.1 Analytical solution methods for $S \subset \mathbb{R}^n$

- *Gradient descent / ascend.* The gradient (a generalization of the one-dimensional derivative) is the direction of the steepest ascend; hence, if a local maximum ("mountain top") is reached, the gradient vanishes. Thus starting at a certain point in the landscape, the gradient leads one to the next local maximum. On the other hand, the negative of the gradient points to the direction of the steepest descend, and following it leads to a local minimum. However, this method can only be applied if the objective function is differentiable (i.e., the landscape is "smooth" and there are no "gorges," "steps," or "peaks"). Moreover, it may lead to a local, but not a global optimum.

- *Newton's method.* If the objective function is even twice differentiable, the Newton method may be applied. It "linearizes" the gradient and leads to the next local optimum. It is very computation-intensive because it involves matrix inversion (of the "Hesse matrix", a generalization of the second derivative).

- *Lagrangian multiplier.* If the optimization problem underlies some constraints, and the objective function is differentiable, the Lagrangian multiplier method can be applied. It is widely used in physics and engineering [39, §14].

- *Simplex algorithm.* The simplex algorithm computes the unique optimum of a linear optimization problem, i.e., both the objective function and all constraints are linear.

6.3.2 Combinatorical solution methods for $S \subset \mathbb{Z}^n$

- *Greedy heuristics.* A greedy heuristic, or a greedy algorithm, is an iterative algorithm where in each iteration step a partial solution is constructed by using the best of the preceding partial solutions. For the TSP, for example, a greedy algorithm is to choose

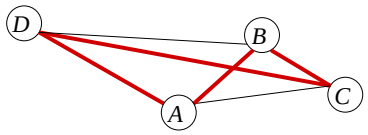


Figure 6.5: A greedy solution of a TSP, starting at *A*. Obviously, the path *A*–*C*–*B*–*D*–*A* is shorter.

in each step the next city to be visited as the one nearest to the current visited city. A solution then could look like as in Figure 6.5, i.e., a greedy algorithm does not guarantee to succeed. Examples of greedy algorithms which are guaranteed to work correctly are Dijkstra's algorithm to find a certain class of shortest path in a network, or the Huffman coding algorithm.

- *Dynamical programming.* Dynamic programming is a technique where in each iteration step a bigger subproblem is constructed by using an optimal subproblem of the last step. It is a cleverly arranged exhaustion without redundant computations. Examples of Dynamic programming algorithms are the Floyd-Warshall algorithm for shortest paths in a directed graph, the Wagner-Whitin method for optimizing the economic order quantity [40, §D.3], or the standard solution of the knapsack problem. There also exists a dynamic programming solution for the TSP.

6.3.3 Biologically inspired solution methods

Various optimization techniques are influenced by principles biological systems, basing on the observation that Nature always finds optimum solutions (although strictly speaking it does not seem to seek one). These techniques all have in common that they use an ensemble of independent individuals which communicate with each other.

- *Artificial neural networks.* Such systems represents a network of simple processing elements called neurons which can exhibit complex global behavior, determined by the connections between themselves and element parameters. They are inspired by the information processing of the brain.

- *Evolutionary algorithms.* Evolutionary algorithms are methods oriented at biological evolution utilizing reproduction, mutation, recombination (crossover), and natural selection ("survival of the fittest") of individuals in a population. Special classes are *genetic algorithms* where individuals are elements of the (usually discrete) search space, *evolution strategy* where similarly individuals are elements of a real search space and the mutation is adapting to certain criteria of the current population, and *evolutionary programming* where individuals are computer programs with varying parameters to optimize the problem.

- *Computational swarm intelligence.* Swarm intelligence methods are designed to find an optimum by the collective behavior of decentralized individual agents communicating

with each other. Examples are *ant colony optimization* for discrete problems, where each ant walks randomly and leaves slowly evaporating pheromones on its way influencing other ants, and *particle swarm optimization* where particles fly through hyperspace having a memory both of their own best position and of the entire swarm's best position and communicating either to neighbor particles or to all particles of the swarm.[3]

[3] *http://jswarm-pso.sourceforge.net*

Chapter 7

Graphs and shortest paths

7.1 Basic definitions

A *graph* represents a set of objects and their relations. Some examples:

Objects	**Relations**
persons	A knows B
players in tennis championship	A plays against B
towns	there exists a highway between A and B
positions in a game of chess	position A transforms to B in one move

The denotation "graph" stems from the usual graphical representation: Objects are represented by vertices[1] and relations by edges. A relation normally is a *directed* edge, an arrow. If the relation is *symmetric*, it is represented by an *undirected* edge. Correspondingly we will consider directed and undirected graphs.

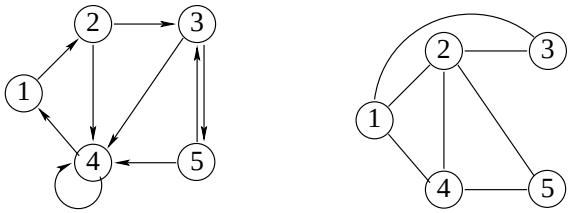


Figure 7.1: Directed and undirected graphs.

Definition 7.1. A *directed graph* (or *digraph*) is a pair $G = (V, E)$ where

1. V is a finite nonempty set whose elements are called *vertices*;
2. A set $E \subseteq V \times V$ whose elements are called *edges*. E is called the *relation*.

□

An edge thus is a pair of vertices. The edge $e = (v, w)$ thus is represented as $v \longrightarrow w$. In this way we recognize the first graph in figure 7.1 as a directed graph. V and E are given by

$$V = \{1,2,3,4,5\}, \qquad E = \{(1,2),(2,3),(2,4),(3,4),(3,5),(4,1),(4,4),(5,3)\}$$

We shortly list some general properties of graphs [30, §2].

[1] The singular form is *vertex*.

- The edge (v,w) is different from the edge (w,v).
- The edge $e=(v,v)$ is possible (depending on E, of course), it is a *self-loop*.
- If $(v,w)\in E$, the vertices v and w are called *adjacent*[2] or *neigbours*
- A *path*, also called a *walk*, is a sequence of vertices $p=(v_0,\dots,v_k)$ such that $(v_i,v_{i+1})\in E$ for $0\leqq i\leqq k-1$. Its *length* is the numbers of the edges, i.e. length = k. A path is called *simple* if no edges are repeated [30, §3.1].
- The maximum number of edges $e=|E|$ in a general graph without parallel edges, consisting of $n=|V|$ vertices, is given if all vertices are joint directly to each other.

  (a) An undirected graph without self-loops can have at most $\binom{n}{2}$ pairs, and thus

$$|E|\leqq\binom{n}{2}. \tag{7.1}$$

  (b) An undirected graph with self-loops can have at most $\binom{n}{2}$ pairs plus n self-loops. Since $n+\binom{n}{2}=n+\frac{n(n-1)}{2}=\frac{(n+1)n}{2}$, we have

$$|E|\leqq\binom{n+1}{2}. \tag{7.2}$$

  (c) A directed graph containing no self-loops: Since there are $\binom{n}{2}$ pairs and each pair can have two directions, a directed graph can have at most $2\binom{n}{2}$ edges, or

$$|E|\leqq 2\cdot\binom{n}{2}=n(n-1) \tag{7.3}$$

  (d) A directed graph containing self-loops can have at most $n(n-1)+n=n^2$ edges, i.e.,

$$|E|\leqq n^2. \tag{7.4}$$

7.2 Representation of graphs

How can a graph be represented in a computer? There are mainly three methods commonly used [26, §4.1.8], two of which we will focus here. Let be $n=|V|$ the number of vertices, and let be $V=\{v_1,\dots,v_n\}$.

1. *(Adjacency matrix)* The *adjacency matrix* $A=(a_{ij})$ of the graph $G=(V,E)$ is an $(n\times n)$-matrix defined by

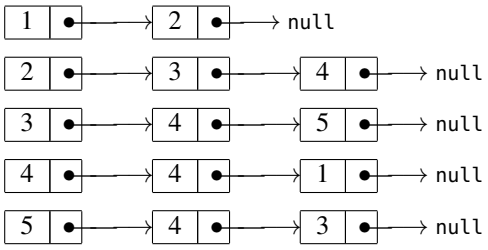
$$a_{ij}=\begin{cases}1 & \text{if } (v_i,v_j)\in E,\\ 0 & \text{otherwise.}\end{cases} \tag{7.5}$$

   For the left graph in figure 7.1 we have $v_i=i$ $(i=1,\dots,5)$. The adjacency matrix therefore is the (5×5)-matrix

$$A=\begin{pmatrix}0&1&0&0&0\\0&0&1&1&0\\0&0&0&1&1\\1&0&0&1&0\\0&0&1&1&0\end{pmatrix}. \tag{7.6}$$

[2] *adjacent*: angrenzend

2. *(Adjacency list)* In an *adjacency list* each vertex has a list of all his neihbors. Via an array $v[]$ of length $n = |V|$ each list is randomly accessible. For the left graph in figure 7.1 we therefore have



There are more possibilities to represent graphs, but adjacency matrices and adcjacency lists are the most important. Both normally are used as *static* data structures, i.e., they are constructed at the start and won't be changed in the sequel. Updates (*insert* and *delete*) play a minor part as compared to the dynamic data structures we have studied so far.

7.2.1 Adjacency matrices contra adjacency lists

An advantage representing a graph by an adjacency matrix is the possibility to check in running time $O(1)$, whether there exist an edge from v_i to v_j, i.e. whether $(v_i, v_j) \in E$. (The reason is simply because one only has to look at the entry (i, j) of the matrix.) A disadvantage is the great requirement of memory storage of seize $O(n^2)$. In particular, if the number of edges is small compared to the number of vertices, memory is wasted. Moreover, the running time for the initialization of the matrix is $\Theta(n^2)$.

The advantage of this representation is the small effort of memory storage of $O(n+e)$ for $n = |V|$ and $e = |V|$. All neighbors are achieved in linear time $O(n)$. However, a test whether two vertices v_i and v_j are adjacent cannot be done in constant running time, because the whole list of v_i must be run through to check the existence of v_j. In worst case the whole adjacency list of vertex v_i has to be run through.

Thus an adjacency matrix should only be chosen, if the intended algorithms include many tests on the existence of edges or if the number of edges is much larger than the number of vertices, $e \gg n$.

	Aadjacency matrix	**Adjacency list**
memory	$O(n^2)$	$O(n+e)$
running time for "$(v_i, v_j) \in E$?"	$O(1)$	$O(n)$
appropriate if	$e \gg n$	$e \lessapprox n$
	many edge searches	few edge searches

7.3 Traversing graphs

7.3.1 Breadth-first search

A first problem we will tackle and on whose solution other graph algorithms base upon is to look systematically for all vertices of a graph. For instance, consider the graph in figure 7.2. The *breadth-first search (BFS)* algorithm proceeds as follows. Find all neighbors of a vertex s; for each neighbor find all neighbors, and so forth. Thus the search discovers the neighborhood of s, yielding a so-called "connected component." To have a criterion whether all vertices in this neighborhood are already discovered, we mark the vertices which are already visited. We assume that we start with white vertices which are colored black when visited during the

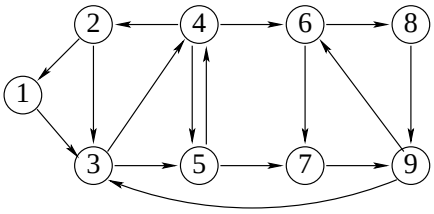
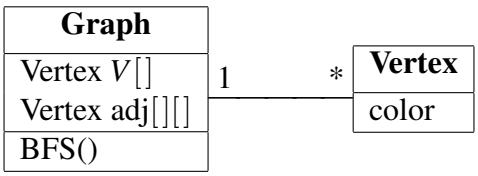


Figure 7.2: A graph to which breadth-first is applied.

execution of BFS. Since we successively examine neighbors of visited vertices, a queue is the appropriate choice as the data structure for subsequently storing the neighbors to be visited next.

Let now $V = \{V[0], V[1], \ldots, V[n-1]\}$ be a set of m vertices. Then each number $s \in \{0, \ldots, n-1\}$ corresponds to a vertex $V[s]$. A graph G is an object which consists of m vertices. This is illustrated by the following class diagram:



The graph contains the set V of vertices and the adjacency list adj where adj[i][j] means that vertex $V[j]$ is in the neighborhood of $V[i]$. Each vertex has a color. The method BFS(s) is the implementation of the following algorithm. For the vertex s it blacks each neighbor of vertex $V[s]$ in graph G. (It is called by G.BFS(s)) BFS has as a local variable a queue $q[]$ of integers containing the indices i of $V[i]$.

```
algorithm BFS (int s)
// *  visits systematically each neighbor of s in graph G = (V,E) and
// *  colors it black. The m vertices are V[i], i = 0,1,...,m-1
q[].empty(); q[].enqueue(s);        // initialize queue and append s
if ( V[s].color == white )
    while (not q[].isEmpty() )
        k ← q[].dequeue();
        if (V[k].color = white)
            V[k].color ← black;
            for (i ← 0; i ≦ adj[k].length; i++;);
                q[].enqueue(i);
```

Complexity analysis. Initializing colors, distances and predecessors costs running time $O(|V|)$. Each vertex is put into the queue at most once. Thus the while-loop is carried out $|E|$ times. This results in a running time

$$T_{\text{BFS}}(|V|,|E|) = O(|V|+|E|). \tag{7.7}$$

Since all of the nodes of a level must be saved until their child nodes in the next level have been generated, the space complexity is proportional to the number of nodes at the deepest level, i.e.,

$$S_{\text{BFS}}(|V|,|E|) = O(|V|+|E|). \tag{7.8}$$

In fact, in he worst case the graph has a depth of 1 and all vertices must be stored.

7.3.2 Depth-first search

Analogously to BFS, *depth-first search (DFS)* visits all vertices reachable from a given vertex $V[s]$. But different to BFS, DFS proceeds going deeper into the graph by each step, see Figure 7.3. It can be formulated recursively, for DFS is carried out on a vertex by carrying out DFS on

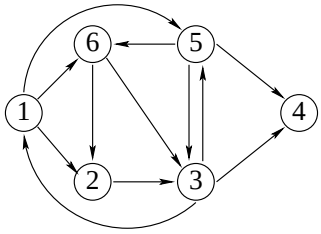


Figure 7.3: A graph to which depth-first search is applied.

all its neighbors and on their neighbors and so forth.

Again, the visited vertices must be marked in order not to be visited twice. DFS is carried out on all vertices of the graph. So it is best represented by defining a subalgorithm *DFS* which is called for each white vertex.

```
algorithm DFS(x)
  //* visits each unmarked vertex reachable from x in graph G and marks it.*/
  if (!x.isMarked())
    x.mark();
    for (y : x.adj)
      DFS(y);
```

Complexity analysis. We observe that DFS is called from DFS maximally $|E|$ times, and from the main algorithm at most $|V|$ times. Moreover it uses space only to store the stack of vertices, i.e.

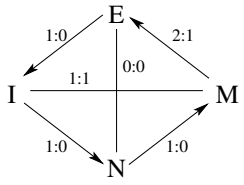
$$T_{\mathrm{DFS}}(|V|,|E|) = O(|V|+|E|), \qquad S_{\mathrm{DFS}}(|V|) = O(|V|). \tag{7.9}$$

7.4 Cycles

Definition 7.2. A closed path $(v_0,\ldots,v_k,v_0)$ i.e., a path where the final vertex coincides with the start vertex, is called a *cycle*. A cycle which visits each vertex exactly once is called a *Hamiltonian cycle*. A graph without cycles is called *acyclic*. □

Example 7.3. In the group stage of the final tournament of the FIFA World Cup, soccer teams compete within eight groups of four teams each. Each group plays a round-robin tournament, resulting in $\binom{4}{2} = 6$ matches in total. A match can be represented by two vertices standing for the two teams, and a directed edge between them pointing to the loser of the match or, in case of a draw, an undirected edge connecting them. For instance, for group E during the world cup 1994 in the USA, consisting of the teams of Ireland (E), Italy (I), Mexico (M), and Norway (N), we have the graph given in Figure 7.4. This group is the only group in World Cup history so far in which all four teams finished on the same points. □

Group E		
Ireland	– Italy	1 – 0
Norway	– Mexico	1 – 0
Italy	– Norway	1 – 0
Mexico	– Ireland	2 – 1
Italy	– Mexico	1 – 1
Ireland	– Norway	0 – 0



Team	Goals	Pts
Mexico (M)	3:3	4
Ireland (E)	2:2	4
Italy (I)	2:2	4
Norway (N)	1:1	4

Figure 7.4: A cycle in the graph representing the match results of group E during the World Cup 1994

7.4.1 Hamiltonian cycle problem HC

A *Hamiltonian cycle* is a cycle in which each vertex of the graph is visited exactly once. The *Hamiltonian cycle problem* (HC)) is to determine whether a given graph contains a Hamilton cycle or not. It is a *decision problem*, not an optimization problem, since it expects the answer "yes" or "no" but not a quantity.

Let X be the set of all possible cycles beginning in vertex 1, i.e., $x = (x_0, x_1, \ldots, x_{n-1}, x_n)$ where $x_0 = x_n = 1$ and where $(x_1, \ldots, x_{n-1})$ is a permutation of the $(n-1)$ vertices $x_j \neq 1$. In other words, X contains all possible Hamiltonian cycles which *could* be formed with the n vertices of the graph. Then a simple algorithm to solve the problem is to perform a "brute force" search through the space X and to test each possible solution x by querying the "oracle function"

$$\omega(x) = \begin{cases} 1 & \text{if } x \text{ is a closed path,} \\ 0 & \text{otherwise.} \end{cases} \tag{7.10}$$

If the graph does not contain a Hamilton cycle, then $\omega(x) = 0$ for all $x \in X$. The oracle function only has to check whether each pair (x_{j-1}, x_j) of a specific possible Hamiltonian cycle is an edge of the graph, which requires time complexity $O(n^2)$ since $|E| \leqq n^2$; because there are n pairs to be checked in this way, the oracle works with total time complexity $T_\omega(n) = O(n^3)$ per query. Its space complexity is $S_\omega(n) = O(\log_2 n)$ bits, because it uses E and x as input and thus needs to store temporarily only the two vertices of the considered edge, requiring $O(\log_2 n)$. In total this gives a time complexity $T_{\text{HC-bf}}$ and a space complexity $T_{\text{HC-bf}}$ of the brute force algorithm of

$$T_{\text{HC-bf}}(n) = O(n^{n+3}) = O(2^{n \log_2 (n+3)}), \qquad S_{\text{HC-bf}}(n) = O(\log n) \tag{7.11}$$

since there are $(n-1)! = O(n^n) = O(2^{n \log_2 n})$ possible permutation ("orderings") of the n vertices. Remarkably, there is no algorithm known to date which is essentially faster. If you find one, publish it and get US$ 1 million.[3] However, for some special classes of graphs the situation is different. For instance, according to a mathematical theorem of Dirac, any graph in which each vertex has at least $n/2$ incident edges has a Hamiltonian cycle. This and some more such *sufficient* criterions are listed in [9, §8.1].

7.4.2 Euler cycle problem EC

A problem being apparently similar to the Hamiltonian cycle problem is the Euler cycle problem. Its historical origin is the problem of the "Seven Bridges of Königsberg", solved by Leonhard Euler in 1736.

An *Euler cycle* is a closed-up sequence of edges, in which each edge of the graph is visited exactly once, If we shortly denote $(x_0, x_1, \ldots, x_m)$ with $x_0 = x_m = 1$ for a cycle, then a necessary condiiton to be Eulerian is that $m = |E|$.

[3] Millenium Prize Problems, Clay Mathematics Institute, 1. "P vs NP" (*www.claymath.org/millennium/P_vs_NP/*). To date, one of the seven problems is solved, Perelman proved the Poincaré conjecture in 2003 but rejected the award.

The *Euler cycle problem* (EC) then is to determine whether a given graph contains an Euler cycle or not. By Euler's theorem [9, §0.8], [26, §1.3.23], a connected graph contains an Euler cycle if and only if every vertex has an even number of edges incident upon it. Thus EC is decidable in $O(n^3)$ computational steps, counting for each of the n vertices x_j in how many of the at most $\binom{n}{2}$ edges (x_j, y) or $(y, x_j) \in E$ it is contained:

$$T_{\text{Euler}}(n) = \Theta(n^3). \tag{7.12}$$

7.5 Shortest paths

We now consider a so-called "weighted graph" which assigns to each edge a certain "length" or "cost." Consider for instance 7.5. Here the weights may have quite different meanings: They

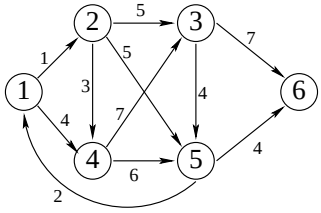


Figure 7.5: A weighted graph.

can express ...

- the distances (i.e. "It has 3 km from 2 to 4.");
- the costs ("It costs €3 from 2 to 4.")
- capacities ("The network bandwidth is 3 MBit per second on the cable from 2 to 4.")
- traveling duration ("It lasts 3 hours from 2 to 4.")

There are many more applications of weighted graphs. We extend our definition of graphs to include the weights.

Definition 7.4. A *weighted graph* $G_\gamma = (V, E, \gamma)$ is a graph $G = (V, E)$ with the *weight*

$$\gamma : E \to \mathbb{R}$$

which assigns a real number to each edge. We often will simply write G for G_γ. □

For an edge $(v, w) \in E$ the weight is thus given by $\gamma(v, w)$.[4] The unweighted graphs which we have seen so far can be considered as special weighted graphs where all weights are constantly 1: $\gamma(v, w) = 1$ for all $(v, w) \in E$.

It is often convenient to write γ as a matrix, where $\gamma_{vw} = \gamma(v, w)$ denotes the weight of the edge (v, w), where the weight is ∞ if the edge (v, w) does not exist; for convenience, such entries are often left blank or are marked with a bar "–". For the weighted graph in Fig. 7.5 we thus obtain the weight matrix

$$\gamma(v,w) = \gamma_{vw} = \begin{pmatrix} - & 1 & - & 4 & - & - \\ - & - & 5 & 3 & 5 & - \\ - & - & - & - & 4 & 7 \\ - & - & 7 & - & 6 & - \\ 2 & - & - & - & - & 4 \\ - & - & - & - & - & - \end{pmatrix} = \begin{pmatrix} \infty & 1 & \infty & 4 & \infty & \infty \\ \infty & \infty & 5 & 3 & 5 & \infty \\ \infty & \infty & \infty & \infty & 4 & 7 \\ \infty & \infty & 7 & \infty & 6 & \infty \\ 2 & \infty & \infty & \infty & \infty & 4 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}. \tag{7.13}$$

[4] Note that we write for short $\gamma(v, w)$ instead of $\gamma((v, w))$.

In this way, the weight matrix is a generalization of the adjacency matrix. With the weight γ we can define the length of a path in graph G_γ.

Definition 7.5. Let be $p = (v_0, v_1, \ldots, v_n)$ be a path in a weighted graph G_γ. Then the *weighted length* of p is defined as the sum of its weighted edges:

$$\gamma(p) = \sum_{i=1}^{n} \gamma(v_{i-1}, v_i). \tag{7.14}$$

A *shortest path* from v to w is a path p of minimum weighted length starting at vertex v and ending at w. This minimum length is called the *distance*

$$\delta(v,w). \tag{7.15}$$

If there exists no path between two vertices v,w, we define $\delta(v,w) = \infty$. □

7.5.1 Shortest paths

We will consider the so-called *single-source shortest path* problem: Given the source vertex $s \in V$, what is a shortest path from s to all other vertices $v \in V$? In principle, this also gives a solution of the following shortest-path problems.

- *(single-destination)* What is a shortest path between an arbitrary vertex to a fixed destination vertex? This problem is a kind of reflexion of the single-source shortest path problem (exchange s and v).
- *(single-pair)* Given a pair $v,w \in V$ of vertices, what is a shortest path from v to w? This problem is solved by running a single-source algorithm for v and selecting a solution containing w.
- *(all-pairs)* What is a shortest path between two arbitrary vertices?

Some algorithms can deal with negative weights. This case poses a special problem. Look at the weighted graph in Fig. 7.6. On the way from v to w we can walk through the cycle to

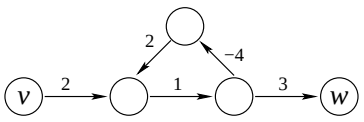


Figure 7.6: A weighted graph with negative weight.

decrease the distance arbitrarily. Hence either the minimum distance between two points that are reachable via a *negative cycle* have distance $-\infty$ — or the problem should be reformulated. Moreover, for an edge $(v,w) \in E$ in a graph $G_\gamma = (V,E,\gamma)$ with only non-negative weights, we simply have

$$\delta(v,w) = \gamma(v,w) \tag{7.16}$$

7.5.2 The principle of relaxation

Theorem 7.6. (Triangle inequality) *Let be $G_\gamma = (V,E,\gamma)$, and $(v,w) \in E$. Denote $\delta(v,w)$ the minimum length between v and w for v, $w \in V$. Then for any three vertices $u,v,w \in V$ we have*

$$\delta(v,w) \leqq \delta(v,u) + \delta(u,w). \tag{7.17}$$

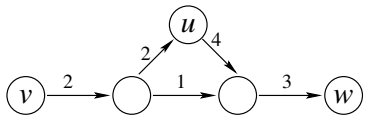


Figure 7.7: Triangle inequality.

Proof. The shortest path from v to w cannot be longer than going via u. □

Note that this inequality holds even if there does not exist a path between one of the vertex pairs (for then $\delta(\cdot,\cdot) = \infty$), or if there are negative weights (for then $\delta(\cdot,\cdot) = -\infty$, and the shortest path already goes via $v \ldots$).

In most shortest path algorithms the *principle of relaxation* is used. It is based on the triangle inequality (7.17):

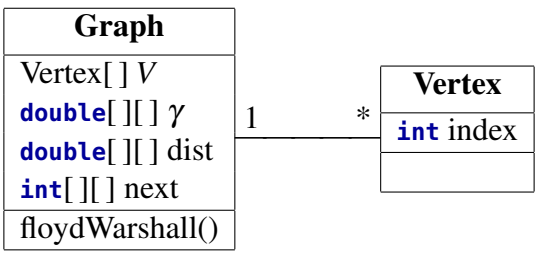
$$\begin{array}{l} \textit{if}\ (\ \mathrm{dist}[v,w] > \mathrm{dist}[v,u] + \mathrm{dist}(u,w)\ )\ \{ \\ \quad \mathrm{dist}[v,w] \leftarrow \mathrm{dist}[v,u] + \mathrm{dist}(u,w); \qquad \mathrm{next}[v,w] \leftarrow u; \\ \} \end{array}$$

Here the matrix entry dist[v][w] stores the information of the minimum distance between v and w, and the matrix entry next[v][w] represents the vertex one must travel through if one intends to take the shortest path from v to w. From the point of view of data structures, they are attributes of an object "vertex." This will be implemented consequently in the Dijkstra algorithm below. The Floyd-Warshall algorithm implements them as attributes *of the graph*; therefore they are given as *two*-dimensional arrays (matrices!)

7.5.3 Floyd-Warshall algorithm

We now consider the Floyd-Warshall algorithm which is fascinating in its simplicity. It solves the all-pairs shortest paths problem. It has been developed independently from each other by R.W. Floyd and S. Warshall in 1962.

Let the vertex be an object as an element of the graph G_γ as given by the following diagram.



(Note that the weight γ and the distance dist are given as two-dimensional arrays.) Then the Floyd-Warshall algorithm is called without a parameter.

```
algorithm FloydWarshall()
//*  Determines all-pairs shortest paths. The n vertices are V[i], i = 0,1,...,n-1
for (v ← 0; v < n; v++)          // initialize
    for (w ← 0; w < n; w++)
        dist[v][w] ← γ[v][w];    next[v][w] ← −1;

for (u ← 0; u < n; u++)
    for (v ← 0; v < n; v++)
        for (w ← 0; w < n; w++)
            // relax:
            if (dist[v][w] > dist[v][u] + dist[u][w])
                dist[v][w] ← dist[v][u] + dist[u][w];    next[v][w] ← u;
```

Unfortunately, the simplicity of an algorithm does not guarantee its correctness. For instance, it can be immediately checked that it does not work for a graph containing a negative cycle. However, we can prove the correctness by the following theorem.

Theorem 7.7 (Correctness of the Floyd-Warshall algorithm)**.** *If the weighted graph $G_\gamma = (V, E, \gamma)$ with $V = \{V[0], V[1], \ldots, V[n-1]\}$ does not contain negative cycles, the Floyd-Warshall algorithm computes all-pairs shortest paths in G_γ. For each index $v, w \in \{0, 1, \ldots, m-1\}$ it yields*

$$\mathrm{dist}[v][w] = \delta(V[v], V[w]).$$

Proof. Because the relaxation goes through all possible edges starting at $V[v]$, in the first two loops we simply have $\mathrm{dist}[v][w] = \gamma[v][w]$. It represents the weights of all paths containing only two vertices.

In each next loop the value of u controls the number of vertices in the paths to be considered: For fixed u all possible paths $p = (e_0, e_1, \ldots, e_u)$ connecting each pair $e_0 = V[v]$ with $e_u = V[w]$ are checked. Since there are no negative cycles, we have

$$u \leqq n-1,$$

because for a shortest path in a graph without negative cycles no vertex will be visited twice. Therefore, eventually we have $\mathrm{dist}[v][w] = \delta(V[v], V[w])$. □

This elegant algorithm is derived from a common principle used in the area of dynamic programming[5], a subbranch of Operations Research [11]. It is formulated as follows.

Bellman's Optimality Principle. *An optimum decision sequence has the property that — independently from the initial state and the first decisions already made — the remaining decisions starting from the achieved (and possibly non-optimum) state yield an optimum subsequence of decisions to the final state.*

An equivalent formulation goes as follows. *An* optimum policy *has the property that — independently from the initial state and the first decisions already made — the remaining decisions yield an optimum policy with respect to the achieved (and possibly non-optimum) state.*

Thus if one starts correctly, Bellman's principle leads to the optimum path.

[5] in German: Dynamische Optimierung

Complexity analysis. The Floyd-Warshall algorithm consists of two cascading loops, the first one running $n^2 = |V|^2$ times, the second one running at most n^3 times [26, §6.1.23]:

$$T_{\mathrm{FW}}(|V|) = O(|V|^3). \tag{7.18}$$

In a "dense" graph where almost all vertices are connected directly to each other, we achieve approximately the maximum possible number of edges $e = O(n^2)$, cf. (7.3). Here the Floyd-Warshall algorithm is comparably efficient. However, if the number of edges is considerably smaller, the three loops are wasting running time.

7.5.4 Dijkstra algorithm

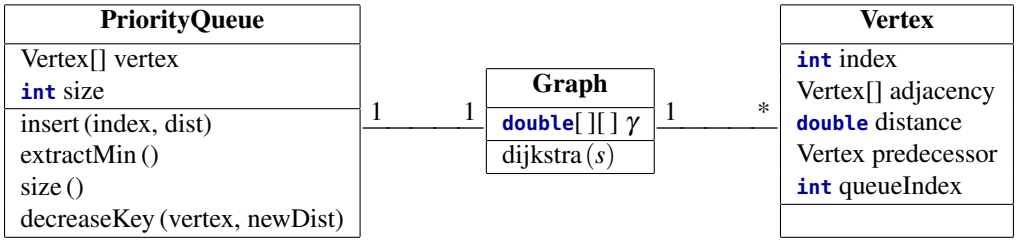
We now will study an efficient algorithm that solves the single-source shortest path problem. It thus answers the question: What are the shortest paths from a fixed vertex to all other vertices? This algorithm was developed by E.W. Dijkstra in 1959. It works, however, only if there are no negative weights.

Similarly to the Floyd-Warshall algorithm, the Dijkstra algorithm decreases successively a distance array dist[] denoting the distance from the start vertex by relaxation. But now it is not the distance of a special pair of vertices which is relaxed, but the absolutely smallest distance value. To get this value, a heap with the distance as key is used.

Unfortunately the distance may often be changed performing the algorithm. Therefore a minimum heap is used as temporary memory and has to be updated frequently. Including this function we speak of a *priority queue*. It possesses the following methods:

- **insert(int vertex, `int` distance):** Inserts vertex along with its distance form the source into the priority queue and reheaps the queue.
- int **extractMin():** Returns the index of the vertex with the current minimal distance from the source and deletes it from the priority queue.
- `int` **size():** Returns the number of elements in the priority queue.
- **decreaseKey(vertex, newDistance):** If the input newDistance $<$ current distance of the vertex in the queue, the distance is decreased to the new value and the priority queue is reheaped from that position on.

The data structures to implement Dijkstra algorithm thus are as in the following diagram.



Here `adjacency` denotes the adjacency list of the vertex. As usual, a vertex $V[i] \in V$ is determined uniquely by its index i. The algorithm Dijkstra is called with the index s of the source vertex as parameter. It "knows" the priority queue h (i.e., h is already created as an object.) The vertex attribute `queueIndex` will be used by the Dijkstra algorithm to store the current index position of the vertex in the priority queue. The algorithm is shown in Figure 7.8. There are some Java applet animations in the Web, e.g.,

http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml

```
algorithm dijkstra(s)
/** finds shortest paths from source s in the graph with vertices V[i], i = 0,1,...,n-1.*/
   // initialize single source:
   for (int i = 0; i < n; i++) {
      V[i].setPredecessor(null);
      V[i].setDistance(INFINITY);
   }
   Vertex source = V[s];
   source.setDistance(0);
   Vertex[] adj = source.getAdjacency();
   for (int i = 0; i < adj.length; i++) {
      adj[i].setPredecessor(source);
      adj[i].setDistance(weight[source.getIndex()][adj[i].getIndex()]);
   }

   PriorityQueue q = new PriorityQueue(vertices);
   while( q.size() > 0 ) {
      u = q.extractMin();
      for( Vertex v : u.getAdjacency() ) {
         // relax:
         d = u.getDistance() + γ[u.getIndex()][v.getIndex()];
         if (v.getDistance() > d) {
            v.setPredecessor(u);
            q.decreaseKey(v,d); // decrease distance and reorder priority queue
         }
      }
   }
```

Figure 7.8: The Dijkstra algorithm

Algorithmic analysis

For the correctness of the Dijkstra algorithm see e.g. [23, Lemma 5.12].

Theorem 7.8. *The Dijkstra algorithm based on a priority queue realized by a heap computes the single-source shortest paths in a weighted directed graph $G_\gamma = (V,E,\gamma)$ with non-negative weights in maximum running time $T_{\text{Dijkstra}}(|V|,|E|)$ and with space complexity $S_{\text{Dijkstra}}(|V|)$ given by*

$$T_{\text{Dijkstra}}(|V|,|E|) = O(|E|\cdot\log|V|), \qquad S_{\text{Dijkstra}}(|V|) = \Theta(|V|). \tag{7.19}$$

Proof. First we analyze the time complexity of the heap operations. We have $n = |V|$ insert operations, at most n extractMin operations and $e = |E|$ decreaseKey operations.

Initializing the priority queue costs at most $O(e\log n)$ running time, initializing the vertices with their distance and predecessor attributes requires only $O(n)$. Determining the minimum with extractMin costs at most $O(e\log n)$, because it is performed at most e times and the reheap costs $O(\log n)$. The method decreaseKey needs $O(\log n)$ since there are at most $O(n)$ elements in the heap.

To calculate the space requirement $S(n)$ we only have to notice that the algorithm itself needs the attributes dist and pred, each of length $O(n)$, as well as the priority queue requiring two arrays of length n to store the vertices and their intermediate minimum distances from the source, plus a single integer to store its current length. In total, this gives $S(n) = O(n)$, and even $S(n) = \Theta(n)$ since this is the minimum space requirement. Q.E.D.

We remark that the Dijkstra algorithm can be improved, if we use a so-called *Fibonacci heap*. Then we have complexity $O(|E|+|V|\log|V|)$ [23, §§5.4 & 5.5].

Chapter 8

Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.[1] Divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. In such a context, a divide-and-conquer algorithm would do more work than necessary, repeatedly solving the common subsubproblems. A dynamic programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

Dynamic programming is typically applied to *optimization problems*. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of the optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required.

8.1 An optimum-path problem

To clarify the basic notions of dynamic programming, we consider a simple example. Suppose the path network graph in figure 8.1, with the costs of each edge ("subpath") being indicated. We are searching for a path from A to O minimizing the costs. Such a path is called an *optimum path*.

The problem can be solved by *exhaustion*, i.e., by computation of all possible paths from A to O. However, the method of dynamic programming provides essential simplifications.

In figure 8.1, the path network is divided into several stages. For instance, the points D, E, and F belong to stage 2. The points are also called *states*. Hence at stage 2, the system is either in state D, E, or F. From a state in a given stage the system can change to a state in the next

[1] "Programming" in this context refers to a tabular method, not to writing computer code ...

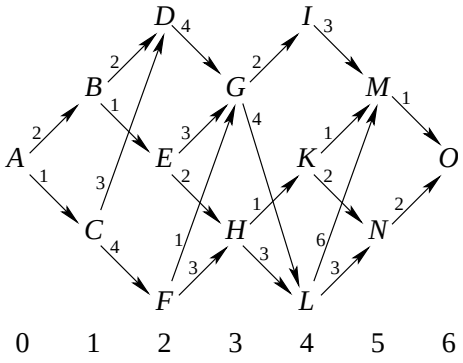


Figure 8.1: An optimum-path problem as an example for dynamic programming.

stage, due to a *decision*. Being in state E at stage 2, the possible decisions are to take either state G or state H at stage 3.

In other words, searching the cost-minimal path from A to O means to search a *decision sequence*, which, starting from the initial state A, yields a state on each stage such that the final state O is reached in a cost-minimal way. At each stage there has to be exactly one state (point) lying on the optimum path. A possible decision sequence is given in table 8.1 and correspondingly by the emphasized path in figure 8.2.

stage	state	decision	cost
0	A	go to C	1
1	C	go to D	3
2	D	go to G	4
3	G	go to L	4
4	L	go to N	3
5	N	go to O	2
6	O		
		total costs	17

Table 8.1: A possible decision sequence for the optimum-path problem.

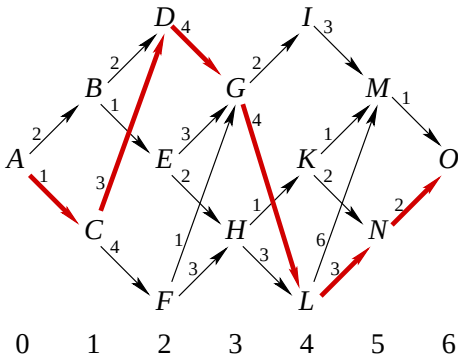


Figure 8.2: A possible decision sequence for the optimum-path problem.

8.1.1 General observations

From the above example we can derive general features of dynamic programming models.

- A dynamic programming problem is divided into $n+1$ stages (or "subproblems"), if there are n decisions to be made.

- At each stage there are several states x_t, exactly one of which at each stage has to be run through by a solution of the problem.
- Being in state x_t at stage t ($t = 0, \ldots, n$), a *decision* (or *action*) a_t has to be made to achieve a state x_{t+1} at stage $t+1$.
- The total decision x consists of a decision sequence $a = (a_0, a_1, \ldots, a_{n-1})$. Here a_t is the decision at stage t, or in other words, the solution of subproblem t. a is also called the *decision vector*.

These connections can be summarized by the formula

$$x_{t+1} = f(x_t, a_t, t), \qquad t = 0, \ldots, n-1. \tag{8.1}$$

Here f is the *transition function*, which changes state x_t at stage t into state x_{t+1} depending on the decision a_t. We call equation (8.1) the *transition law*, or *law of motion*, cf. [2]. It is important to notice that state x_{t+1} at stage $t+1$ *solely* depends on stage t, state x_t, and decision a_t. Other states or actions at stage t have no influence on x_{t+1}.

But what has to be optimized at all? We have to minimize the sum of the costs c_t that are caused by each decision a_t changing from state x_t to x_{t+1}. Formally we write

$$c(x_0, a) \;=\; \sum_{t=0}^{n-1} c_t(x_t, a_t) \;\longrightarrow\; \min_a . \tag{8.2}$$

where $a = (a_0, a_1, \ldots, a_{n-1})$ is the decision vector, and x_t with $t > 0$ results from the decision a_{t-1} and the state x_{t-1}. The function c is called the *(total) cost function*. It is "separable," since it can be separated into the sum of each stage costs, $c(x_0, a) = \sum c_t(x_t, a_t)$. For a state sequence x_s–x_{s+1}– $\ldots$ –x_t, with $0 \leqq s < t < n$, where each state x_k results from a decision according to the recursive transition law (8.1), we will also denote the cost function of by $c(x_s, x_{s+1}, \ldots, x_t)$, and the minimum cost value from state x_s to x_t simply by $c(x_s, x_t)$.

The dynamic programming method now consists of stepwise recursive determinations of optimal subpaths. All optimal subpaths are computed recursively, i.e. by using previously computed optimal subpaths. The recursion relies on the following fundamental principle.

Bellman's Optimality Principle. *An optimum decision sequence has the property that — independently from the initial state and the first decisions already made — the remaining decisions starting from the achieved (and possibly non-optimum) state yield an optimum subsequence of decisions to the final state.*

An equivalent formulation goes as follows. *An* optimum policy *has the property that — independently from the initial state and the first decisions already made — the remaining decisions yield an optimum policy with respect to the achieved (and possibly non-optimum) state.*

For our optimum-path problem the Bellman principle thus means: Each subpath to O, e.g., from D to O, must be an optimum connection from D to O, no matter whether the actual total optimum path from A to O runs through D or not.

8.1.2 Solving the path problem by dynamic programming

We start *at the end* of the path at stage 6 and follow the path from the 5th to the 6th stage. On stage 5 the optimum path only can take the states M and N, i.e. $x_5 \in \{M, n\}$, and the only

possible "decision" $x_5 = O$. Thus we have

$$c_5(M,C) = 1, \qquad c_5(N,C) = 2.$$

These are the optimum costs from M and N, repectively. We write these values directly above them. as in fig. 8.3 (left).

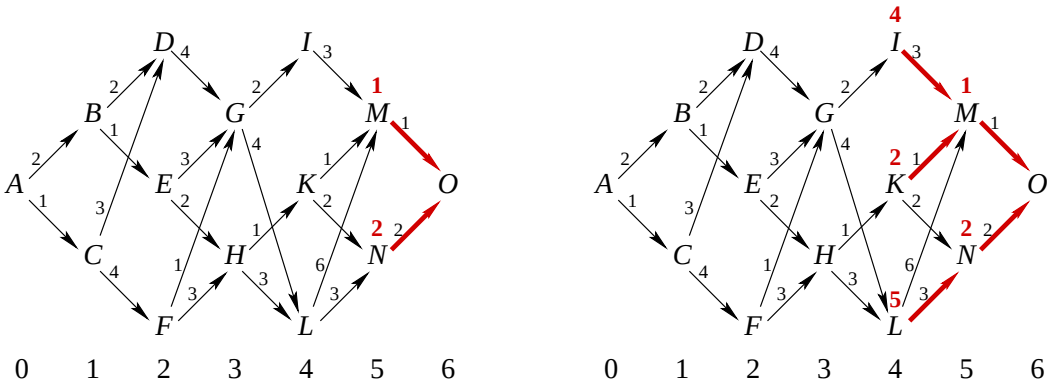


Figure 8.3: The optimum subpaths from stage 5 to stage 6 (left), and from stage 4 to stage 6 (right).

Back at stage 4, there are the possible states I, K, and L which can be achieved by the optimum path. From I there is only one state achievable to reach O, namely M. This yields the optimum costs $c(I,O) = c_4(I,M) + c_5(M,O) = 3+1 = 4$, written above the letter I.

From K there are two possible decisions, K–M or K–N. Hence the total costs from K to O are either $c(K,M,O) = 2$, or $c(K,N,O) = 4$, i.e. the minimal costs from K to O are

$$c(K,O) = \min[c(K,M,O), c(K,N,O)] = \min[2,4] = 2.$$

Analogously,

$$c(L,O) = \min[c(L,M,O), c(L,N,O)] = \min[7,5] = 5.$$

This concludes the computation of the optimum subpaths from stage 4 to stage 6, cf. fig. 8.3 (right).

In a similar manner we achieve recursively the optimum subpaths from stage 3, 2, 1, and 0, resulting in the optimum path A–B–E–H–K–M–O, with total costs $c(A,O) = 8$.

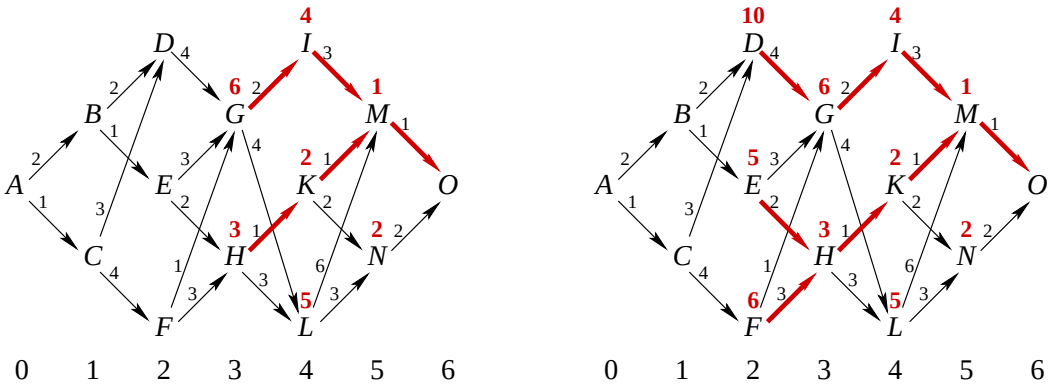


Figure 8.4: The optimum subpaths from stage 3 to stage 6 (left), and from stage 2 to stage 6 (right). Note that by stage 3, L–N–O cannot belong to the total optimum path!

Remark. In the consideration of the transition from stage 3 to stage 4 it is obvious that the cheapest subpath between two stages (here F–G with cost 1) need not necessarily lie on an optimum subpath.

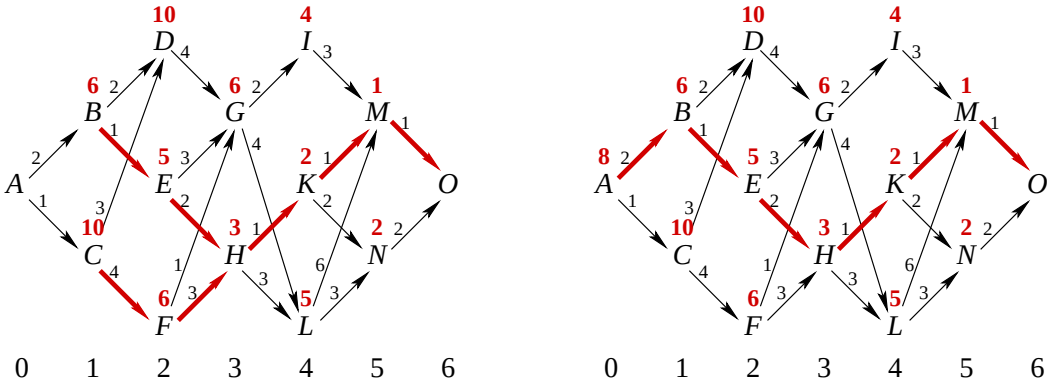


Figure 8.5: The optimum subpaths from stage 1 to stage 6 (left), and the optimum path from A to O (right).

8.2 The Bellman functional equation

To sum up, dynamic programming can be applied to a system which moves through a sequence of states $x_0, x_1, \ldots, x_n$ at the stages (or times) $t = 0, \ldots, n$, where the transition from a state x_t at stage t depends on the decision (or action) a_t and the time t according to the transition law

$$x_{t+1} = f(x_t, a_t, t), \qquad t = 0, \ldots, n-1. \tag{8.3}$$

The problem is to minimize the total costs $c(x_0, a)$ caused by the decision sequence $a = (a_0, \ldots, a_{n-1})$ on the initial state x_0,

$$c(x_0, a) \;=\; \sum_{t=0}^{n-1} c_t(x_t, a_t) \;\longrightarrow\; \min_a. \tag{8.4}$$

Such a problem is called a *sequential* or *multistage decision problem*, where the time variable t (or an arbitrary index i) is used to order the sequence. The dynamic programming approach then consists of solving the recursive equation

$$g_t(x_t) = \min_{a_t \in \mathscr{A}(x_t)} \Big[c_t(x_t, a_t) + g_{t+1}(f(x_t, a_t, t)) \Big] \tag{8.5}$$

for each state x_t at stage t. $\mathscr{A}(x_t)$ is the *decision space*, i.e. the set of all possible transitions (paths) from x_t to a state x_{t+1}. Equation (8.5) is called the *Bellman functional equation*. The minimum value $g_t(x_t)$ thus yields the optimum path from x_t to the end x_n.

This value has to be computed for all possible states x_t. Starting at the end, i.e. with $t = n-1$, one computes successively the optima for all n stages. The optimum path from stage 0 with initial state x_0 to the final state x_n at stage n then is the solution of the problem.

Hence it is natural to treat a sequence of decisions by reversing the order. It is for this reason that the method is also called *backwards induction*.

8.3 Production smoothing

Production smoothing problems occur frequently, if the demand varies over several periods and the production has to be adjusted because of limited storage capacity. In the sequel we will consider a concrete problem, formulate the corresponding dynamic programming problem and solve it according to section 8.1.

8.3.1 The problem

A firm plans the production of four successive periods. The total demand for the produced commodities of $b_{\text{total}} = 90$ qu ("quantity units") distributes on the four periods $t = 0, 1, 2, 3$ as

follows.

$$b_0 = 10 \text{ qu}, \quad b_1 = 20 \text{ qu}, \quad b_2 = 20 \text{ qu}, \quad b_3 = 40 \text{ qu}. \tag{8.6}$$

There are the following restrictions.

- Because the firm has only limited production capacities, it can produce maximally 30 qu per period.
- The production occurs in lots of 10, 20, or 30 qu.
- The firm has fixed production costs of 11 cu ("currency units") per period.
- The variable production costs c_p^{var} per lot are depending on the lot size x:

x	0	10	20	30
$c_p^{\text{var}}(x)$	0	5	11	26

  This yields the total production costs $c_p(x)$ as the sum of fixed and variable costs as

x	0	10	20	30
$c_p(x)$	11	16	22	37

- The storage capacity $l_{\max}$ is only 20 qu per period, $l_{\max} = 20$ qu.
- At the beginning and at the end of the whole planning period no quantities stored are allowed.
- Per qu of stored products emerge 0.2 cu storage costs c_s per period, $c_s(l_t) = 0.2\, l_t$ cu, where l_t is the inventory at period t.
- At the end of each period the demand is called once at a time. During this period there do not emerge any storage costs c_s.

We search for the production quantities of each period that minimize the sum of production and storage costs over the whole planning period,

$$c(l_0, x) = \sum_{t=0}^{3} (c_p(x_t) + c_s(l_t)) \longrightarrow \min_x . \tag{8.7}$$

8.3.2 Reformulation as a sequential decision problem

To solve the problem by dynamic programming, we reformulate it as a sequential decision problem, in terms of stages, states, and decisions.

- The stage t corresponds to the beginning of period t;
- the states correspond to the storage inventory l_t;
- the decision at stage i coincides with the production quantity x_t.

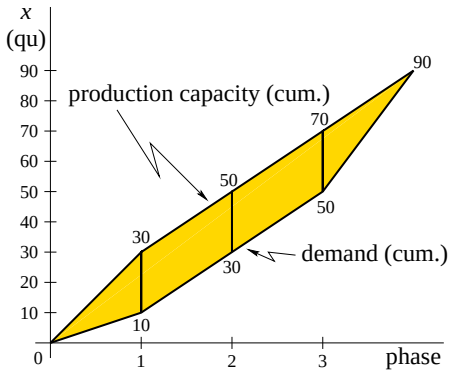


Figure 8.6: The decision space $\mathscr{A} = \bigcup_{l_t} \mathscr{A}(l_t)$ (shaded region) of the production-smoothing problem. The cumulative demand $B(t) = \sum_{i=0}^{*} b_i$ bounds $\mathscr{A}$ from below, whereas the cumulative production capacity $L(t) = t l_{\max}$ bounds it from above.

We now sketch the decision space. It is bounded from below by the requirement that the demand b_t has to be covered (lower limit in fig. 8.6). The production capacity of 30 qu needs not to be regarded since decision space is bounded from above by the storage capacity $l_{\max} = 20$ qu. This yields the decision space as the shaded region in figure 8.6. The decision space is restricted by the demands b_t and the storage capacity $l_{\max}$.

Since the storage inventory l_{t+1} in period $t+1$ depends only on the inventory l_t and the quantity x_t produced in the previous period t (the demand b_t is known and thus a constant parameter), the following relation can be established, $l_{t+1} = f(x_t, l_t)$ with

$$\boxed{f(x_t, l_t) = l_t + x_t - b_t.} \tag{8.8}$$

This is the classical *storage balance equation* of discrete-time production planning, saying that the storage inventory at the beginning of the period t is given by the storage inventory at the beginning of the preceding period, increased by the production quantity x_t and decreased by the demand b_t. In the context of the dynamic programming method this means that state l_{t+1} at stage $t+1$ depends only on the decision x_t in the preceding stage. Figure 8.7 illustrates the problem.

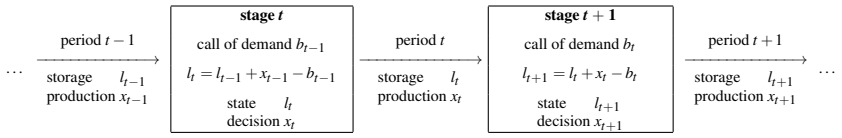


Figure 8.7: Course of production and storage, and the relationship of periods and stages.

The costs of a period t consist of the production costs $c_p(x_t)$ and the storage costs $c_s(l_t)$ of the period,

$$c_t(l_t, x_t) = c_p(x_t) + c_s(l_t) = c_p(x_t) + 0.2\, l_t.$$

According to equ. (8.7) the total cost minimization problem is stated by

$$c(l_0, x) = \sum_{t=0}^{3} c_t(l_t, x_t) \longrightarrow \min_x. \tag{8.9}$$

Let $g_t(l_t)$ be the minimum cost necessary to reach the final sought for storage inventory state l_4, starting with the inventory l_t, without violating one of the constraints $0 \leqq x_t \leqq 30$, $0 \leqq l_t \leqq l_{\max}$. With (8.8) this yields the recursion formula

$$\boxed{g_t(l_t) = \min_{x_t \in \mathscr{A}_t(l_t)} \Big[c_t(l_t, x_t) + g_{t+1}(l_t + x_t - b_t) \Big]} \tag{8.10}$$

with the decision spaces $\mathscr{A}_t(l_t) \subset \{0,10,20,30\}$. The equation says that from the storage inventory l_t at the beginning of period t, the target l_4 is reached at minimum costs by minimizing the sum of production and storage costs to reach l_{t+1} and the minimum costs to reach l_4 from l_{t+1}.

Equation (8.8) is the transition law of our multistage decision problem, i.e. $f(x_t,l_t) = l_t + x_t - b_t$, and (8.10) is its Bellman functional equation.

8.3.3 The graphical solution

To solve the production-smoothing problem, we sketch its decision graph. For this it is convenient first to list the admissible states l_t at each stage t. Since at the beginning the storage has to be empty, we have state $l_0 = 0$ at stage $t = 0$ must be zero, $l_0 = 0$. At stage $t = 1$, the possible production quantities are 0, 10, 20, or 30, but the demand is $b_1 = 10$; this yields the possible storage states $l_1 = 0$, 10, or 20, having produced lots of 10, 20, or 30 qu, respectively. Analogously, we obtain the possible states at stages $t = 2$ and $t = 3$, restricted by the storage

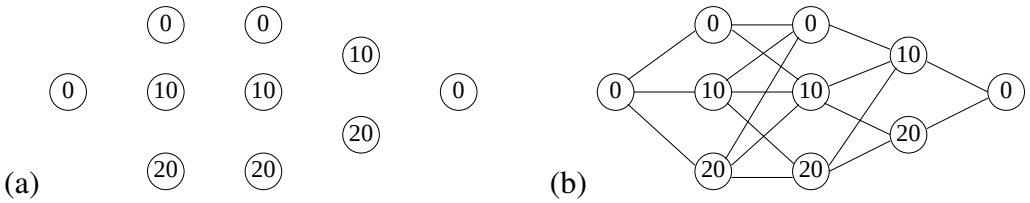


Figure 8.8: (a) The admissible states at each stage, given by the decision space $\mathscr{A} = \bigcup_{l_t} \mathscr{A}(l_t)$ for $t = 0,\ldots,4$. (b) The possible decisions.

capacity of $l_{\max} = 20$ qu; the state $l_4 = 0$ is prescribed by the condition that the storage has to be empty at the end of the period. This yields Fig. 8.8 (a). The possible decisions then are drawn, yielding Fig. 8.8 (b).

In a tedious way we then attach the total cost $c_p(x_t) + c_s(l_t)$, with $c_s(l_t) = 0.2\,l_t$, of each decision x_t. At stage $t = 0$, the storage costs $c_s(l_0)$ are zero, since the storage is empty, $c_s(l_0) = 0$. There are three possible decisions, $x_0 = 10$, 20, or 30, each causing production costs of $c_p(x_0)$ = 16, 22, or 37 cu, respectively. At the next stage $t = 1$, the possible decisions for state $l_1 = 0$ are $x_1 = 20$ or 30, yielding total costs of $c_p(x_1) + c_s(0) = 22$ or 37 cu. For state $l_1 = 10$ we have storage costs $c_s(10) = 2$, and thus the total costs of the three decisions are 16 + 2, 22 + 2, or 37 + 2, respectively. For $l_1 = 20$, the storage costs are $c_s = 4$, i.e. the total costs are 15, 22, or 26. Analogous calculations yield Fig. 8.9 (a). The optimal subpaths are achieved by computing

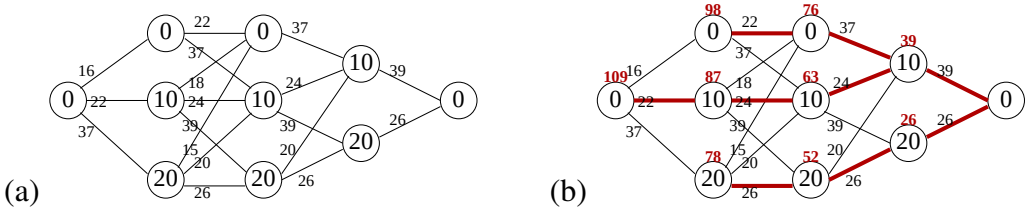


Figure 8.9: (a) The decision graph with the respective costs. (b) The optimal paths from each state to the final state are marked.

backwards and attaching the minimal cost to each state, see Fig. 8.8 (b).

8.3.4 The solution: Wagner-Whitin algorithm

We begin at stage 4 (the end of period 4). Since the state l_4 of stage 4 is the final state $l_4 = 0$ and thus already reached, we have trivially $g_4(0) = 0$.

At stage 3, i.e. the end of period 3, the demand $b_3 = 40$ has to be covered by the decision x_3. By the storage balance equation (8.8) we have $l_4 = l_3 + x_3 - b_3$, or $l_3 + x_3 = 40$. Together with the Bellman equation this yields

$$x_3 = 40 - l_3, \qquad g_3(l_3) = \min_{x_3 \in \{0,10,20,30\}} \Big[c_p(x_3) + 0.2\, l_3 + g_4(l_3 + x_3 - 40) \Big].$$

In the following table the production costs $c_p(x_3)$ for each decision x_3 is listed in the right column, and the storage costs $c_s(l_3) = 0.2\, l_3$ for each storage quantity l_3 in the lowest row. The aim of the first iteration is to list the values of $g_3(l_2 + x_2 - 20)$ for the admissible combinations of x_3 and l_3 (i.e., $x_3 \in \{0,10,20,30\}$ and $0 \leqq l_3 \leqq l_{\max} = 20$). First the values of the squared brackets are computed (note that $g_4(l_4) = 0$). For this purpose we determine the term $0.2\, l_3$ by the following table.

l_3	0	10	20	$c_p(x_3)$
x_3				
0	–	–	–	11
10	–	–	–	16
20	–	–	0	22
30	–	0	–	37
$0.2\, l_3$	0	2	4	

$g_3(0)$ is not defined, because $x_3 = 40$ is not in the decision space, the production capacity is maximally $x_3 = 30$. To obtain the optimum decision x_3 in dependence from the storage quantity l_3, i.e. $x_3(l_3)$, as well as the values $g_3(l_3)$, we have to determine the minimum of each l_3-column (bordered in the following table).

l_3	0	10	20
x_3			
0	–	–	–
10	–	–	–
20	–	–	26
30	–	39	–
$x_3(l_3)$	–	30	20
$g_3(l_3)$	–	39	26

Thus the values of $g_3(l_2 + x_2 - 20)$ for the admissible combinations of x_3 and l_3 are 39 and 26.

Analogously, for stage 2 we have $l_3 = l_2 + x_2 - 20$, and therefore

$$g_2(l_2) = \min_{x_2 \in \{0,10,20,30\}} \Big[c_p(x_2) + 0.2\, l_2 + g_3(l_2 + x_2 - 20) \Big].$$

We obtain the following left table for the values of $g_3(l_2 + x_2 - 20)$ in dependence of l_2 and x_2, viz. 39 or 26. This yields the right table for the optimum decision $x_2(l_2)$ for l_2 and the corresponding values of $g_2(l_2)$.

l_2	0	10	20	$c_p(x_2)$
x_2				
0	–	–	–	11
10	–	–	39	16
20	–	39	26	22
30	39	26	–	37
$0.2\, l_2$	0	2	4	

l_2	0	10	20
x_2			
0	–	–	–
10	–	–	59
20	–	63	52
30	76	65	–
$x_2(l_2)$	30	20	20
$g_2(l_2)$	76	63	52

For stage 1 we have $l_2 = l_1 + x_1 - 20$, and therefore

$$g_1(l_1) = \min_{x_1 \in \{0,10,20,30\}} \Big[c_p(x_1) + 0.2\, l_1 + g_2(l_1 + x_1 - 20) \Big].$$

We obtain the following left table for the values of $g_2(l_1 + x_1 - 20)$ in dependence of l_1 and x_1, and the right table for the optimum decision and the corresponding values of $g_1(l_1)$.

x_1 \ l_1	0	10	20	$c_p(x_1)$
0	–	–	76	11
10	–	76	63	16
20	76	63	52	22
30	63	52	–	37
$0.2\, l_1$	0	2	4	

x_1 \ l_1	0	10	20
0	–	–	91
10	–	94	83
20	**98**	**87**	**78**
30	100	91	–
$x_1(l_1)$	20	20	20
$g_1(l_1)$	98	87	78

Finally, at stage 0 we have $l_1 = l_0 + x_0 - 10$, i.e.

$$g_0(l_0) = \min_{x_0 \in \{0,10,20,30\}} \Big[c_p(x_0) + 0.2\, l_0 + g_1(l_0 + x_0 - 10) \Big].$$

We obtain the following tables for the values of $g_1(l_0 + x_0 - 10)$, as well as for the optimum decision and the corresponding values of $g_0(l_0)$.

x_0 \ l_0	0	10	20	$c_p(x_0)$
0	–	–	–	11
10	98	–	–	16
20	87	–	–	22
30	78	–	–	37
$0.2\, l_1$	0	2	4	

x_1 \ l_1	0	10	20
0	–	–	–
10	114	–	–
20	**109**	–	–
30	115	–	–
$x_1(l_1)$	20	–	–
$g_1(l_1)$	109	–	–

Thus the optimum decision sequence reads in tabular form as follows.

stage t	l_t	x_t	b_t
0	0	20	10
1	10	20	20
2	10	20	20
3	10	30	40
4	0		

8.4 The travelling salesman problem

In the *travelling salesman problem* a salesman must visit n cities. There is a cost c_{ij} to travel from city i to city j, and the salesman wishes to make a tour whose total cost is minimum (where the total cost is the sum of the individual costs between the cities). (Besides: "cost" may be replaced by "distance" or by or "time.")

In the general formulation of the problem we admit the given cost matrix (c_{ij}) to be unsymmetric (it need not hold $c_{ij} = c_{ji}$, i.e. it may be cheaper to travel from city i to j than to come

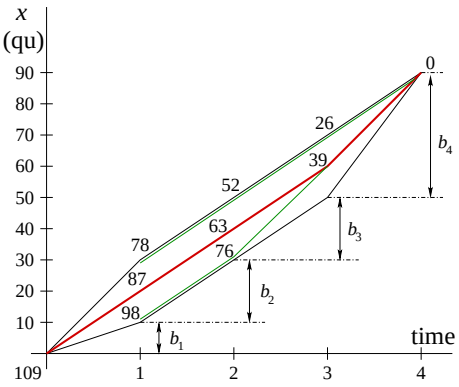


Figure 8.10: The solution of the production-smoothing problem.

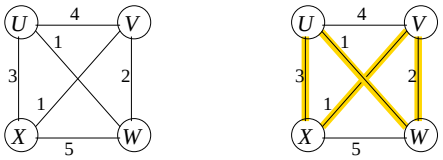


Figure 8.11: A travelling salesman problem and its solution, a minimum-cost tour with cost 7.

from j to i). Moreover, the triangle inequality ($c_{ik} \leqq c_{ij} + c_{jk}$) need not hold as well. Also, the costs c_{ij} may by ∞, which means that there is no direct way from city i to city j.

The travelling salesman problem then is to search for a permutation $\pi : \{1,\ldots,n\} \to \{1,\ldots,n\}$ that minimizes the cost function

$$c(\pi) = \sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)} \; + \; c_{\pi(n),\pi(1)} \;\longrightarrow\; \min. \tag{8.11}$$

Since there may be ∞-entries in the cost matrix (c_{ij}), it is possible that a solution does not exist at all, i.e. that each permutation π leads to an infinite cost value $c(\pi)$.

A naïve algorithm would go through all $n!$ permutations π and compute $c(\pi)$ each time. (It suffices to consider only permutations with $\pi(1) = 1$; there are $(n-1)!$ of those ones.) Therefore, this algorithm has complexity $O(n!)$, even worse than an exponential complexity.

By dynamic programming this naïve ansatz can be improved. However, the resulting algorithm will still have an exponential complexity. It is widely believed that there does not exist an algorithm for the travelling with better complexity at all! This problem is one of a few which are called “NP-complete”.

8.4.1 A solution of the travelling salesman problem

If an optimum round tour starts at city 1 and then visits city k, then the subtour from city k through the cities $\{2, \ldots, n\} \setminus \{k\}$ back to city 1 must be optimal, too. Thus the optimality principle shines through, which we will apply as follows.

For $i \in \{1,\ldots,n\}$ and $S \subset \{1,\ldots,n\}$, let $g(i,S)$ be the length of the shortest path that starts at city i and then visits each city in S exactly once and terminates at city 1. Note that a solution of the travelling salesman problem is given if the length $g(1,\{2,\ldots,n\})$ is computed. The function $g(i,S)$ can be described recursively:

$$g(i,S) = \begin{cases} c_{i1} & \text{if } S = \emptyset, \\ \min\limits_{j \in S} \left[c_{ij} + g(j, S \setminus \{j\}) \right] & \text{if } S \neq \emptyset. \end{cases} \tag{8.12}$$

Our dynamic programming algorithm needs a table for the $g(i,S)$-values, where the combinations with $1 \in S_i$ are not needed. The algorithm works as follows.

```
for ( i=2; i≦n; i++ )  g(i,∅) = c_{i1};
for ( k=1; k≦n−2; k++ ) {
   for ( S, |S|=k, 1∉S ) {
      for ( i∈{2,...,n}\S ) {
         compute g(i,S);
      }
   }
}
compute g(1,{2,3,...,n});
```

The complexity of the algorithm is given by the product

$$(\text{magnitude of the table}) \cdot (\text{complexity of each table entry}).$$

The table magnitude is determined by (number of the i's) $\cdot$ (number of the S's) $\leqq n2^n$. To compute a table entry a loop has to be programmed which searches the minimum of all $j \in S$. This has complexity of $O(n)$. Hence the total complexity is given by

$$T_{\text{TSP}}(n) = O(n^2 2^n) = O(2^{n+2\log_2 n}). \tag{8.13}$$

This is still an exponential complexity, but it is faster than $O(n!)$, cf. §7.4.1.

Example 8.1. Let be given the cost matrix

$$C = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix} \tag{8.14}$$

cf. figure 8.12. The algorithm computes the following values.

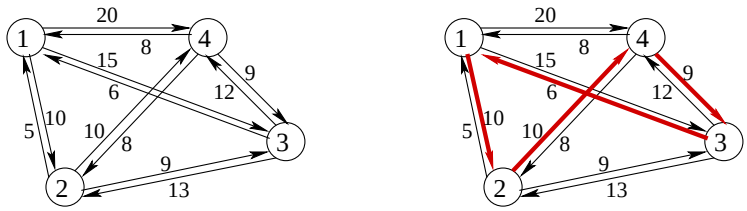


Figure 8.12: A travelling salesman problem and its solution, a minimum-cost tour 1—2—4—3—1 with cost 35.

$$g(2,\emptyset) = c_{21} = 5, \quad g(3,\emptyset) = c_{31} = 6, \quad g(4,\emptyset) = c_{41} = 8,$$

as well as

$$\begin{aligned} g(2,\{3\}) &= c_{23} + g(3,\emptyset) = 15, \quad g(2,\{4\}) = c_{24} + g(4,\emptyset) = 18, \\ g(3,\{2\}) &= c_{32} + g(2,\emptyset) = 18, \quad g(3,\{4\}) = c_{34} + g(4,\emptyset) = 20, \\ g(4,\{2\}) &= c_{42} + g(2,\emptyset) = 13, \quad g(4,\{3\}) = c_{43} + g(3,\emptyset) = 15, \\ g(2,\{3,4\}) &= \min[c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})] = 25, \\ g(3,\{2,4\}) &= \min[c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})] = 25, \\ g(4,\{2,3\}) &= \min[c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})] = 23, \end{aligned}$$

$g(1,\{2,3,4\}) = \min[c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})] = \min[35,40,43] = 35.$

Storing the subpaths that yield the respective solutions, we achieve the optimum tour: it is 1—2—4—3—1. □

Chapter 9

Simplex algorithm

Let us introduce the simplex algorithm by applying it to a concrete optimization problem. Afterwards we will take a look at some of its general properties.

Example 9.1. *(Production scheduling)* A firm gains profit of 2 k€[1] with product 1, and profit of 2.2 k€ with product 2. To produce these products there are two machines A and B available. Machine A can only be used up to 100 hours, and machine B up to 80 hours. (The remaining hours are needed for maintaining.) To produce product 1, machine A is needed 1 hour a week and machine B 2 hours; the respective numbers for product 2 are 2 hours on A and 1 hour a week on B. There are two resource materials R and S needed. R is available only up to 960 kg a week, and material S only 1200 kg a week. Producing product 1 requires 16 kg of R and 20 kg of S, whereas product 2 requires 15 kg of R and 16 kg of S. The production schedule maximizing the profit is searched. □

9.1 Mathematical formulation

1. *(Determination of the objective function)* What is the function that has to be optimized? In the example it is the profit function

$$z = 2x_1 + 2.2x_2, \tag{9.1}$$

where x_1 is the number of instances of product 1 and x_2 the respective number of product 2. It is called the *objective function*[2] of the problem. Observe that quite naturally x_1 and x_2 are nonnegative,

$$x_1, x_2 \geqq 0, \tag{9.2}$$

These inequalities are called *primary constraints*.

2. *(Determination of the constraints)*. Once the quantities x_1 and x_2 are defined, the constraints are directly derived:

$$\begin{array}{rcrcr} x_1 & + & 2x_2 & \leqq & 100 \\ 2x_1 & + & x_2 & \leqq & 80 \\ 16x_1 & + & 15x_2 & \leqq & 960 \\ 20x_1 & + & 16x_2 & \leqq & 1200 \end{array} \tag{9.3}$$

[1] 1 k€ = 1000 €

[2] "objective function" in German: "Zielfunktion"

We can rewrite equations (9.1), (9.2) and (9.6) in matrix notation to reformulate the optimization problem as

$$z = c^* \cdot x \longrightarrow \max, \qquad \text{under the constraints} \quad Ax \leqq b, \quad x \geqq 0, \tag{9.4}$$

where

$$c^* = (2, 2.2), \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ 16 & 15 \\ 20 & 16 \end{pmatrix}, \quad b = \begin{pmatrix} 100 \\ 80 \\ 960 \\ 1200 \end{pmatrix}, \tag{9.5}$$

(Note that c^* is a row vector and denotes the transpose of the column vector c.) The crucial trick of the simplex algorithm now is to introduce some extra *slack variables* y_i to transform the inequalities (9.3) into equalities:

$$\begin{array}{rcrcrcrcl} x_1 & + & 2x_2 & + y_1 & & & & = & 100 \\ 2x_1 & + & x_2 & & + y_2 & & & = & 80 \\ 16x_1 & + & 15x_2 & & & + y_3 & & = & 960 \\ 20x_1 & + & 16x_2 & & & & + y_4 & = & 1200 \end{array} \tag{9.6}$$

It is notationally convenient to record the information content of (9.4) in a so-called *simplex tableau*, as follows.

	x_1	x_2	
z	2	2.2	0
y_1	1	2	100
y_2	2	1	80
y_3	16	15	960
y_4	20	16	1200

(9.7)

9.2 The simplex algorithm in detail

We now write the formulation in a more general manner. Let be given the optimization problem

$$\boxed{z = c^* \cdot x \longrightarrow \max, \qquad \text{under the constraints} \quad Ax \leqq b, \quad x \geqq 0,} \tag{9.8}$$

where

$$c^* = (c_1, c_2, \ldots, c_n), \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}, \tag{9.9}$$

An optimization problem in the form of equation (9.8) is called a *linear optimization problem*. It is linear, because both the objective function z and the constraints depend linearly on x. Methods to solve this problem are put together under the notion of *linear programming* or *linear optimization*.

The simplex tableau of (9.8) is given by

	x_1	$\cdots$	x_n	
z	c_1	$\cdots$	c_n	b_0
y_1				b_1
$\vdots$		A		$\vdots$
y_m				b_m

$=$

	x_1	$\cdots$	x_n	
z	c_1	$\cdots$	c_n	b_0
y_1	a_{11}	$\cdots$	a_{1n}	b_1
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
y_m	a_{m1}	$\cdots$	a_{mn}	b_m

(9.10)

At the start we have $b_0 = 0$. We will call the top row of this tableau the "z-row." The simplex algorithm now consists of the following steps which are repeated until all the entries in the z-row of the tableau are negative.

1. *Determine the pivot column.* The "pivot column" is the column of a maximum *positive* value in the z-row,
$$j_p \in \{j : c_j > 0 \ \wedge \ c_j = \max_k [c_k]\}. \tag{9.11}$$
If there is no positive value c_j, the method is terminated.

2. *Determine the pivot row and the pivot element.* The "pivot row" is the row i_p for which the quotient $b_k/a_{k,j_p}$ with $a_{k,j_p} > 0$ is minimal,
$$i_p \in \left\{ i : a_{i,j_p} > 0 \ \wedge \ a_{i,j_p} = \min_k \left[\frac{b_k}{a_{k,j_p}} \right] \right\}. \tag{9.12}$$
If there is no positive matrix entry $a_{i,j_p} > 0$, the algorithm stops, there does not exist a solution. The matrix entry a_{i_p,j_p} is the *pivot element*.[3] Save the value of the pivot element $d \leftarrow a_{i_p,j_p}$.

3. *Exchange the pivot row and column variables.* Exchange $y_{i_p} \leftrightarrow x_{j_p}$.

4. *Change the z-row values.* The z-row values c_j are changed according to the following cases:
$$c_j \leftarrow \begin{cases} -c_j/d & \text{if } j = j_p, \quad (c_j \text{ is in the pivot column}) \\ c_j - c_{j_p} a_{i_p j}/d & (\text{"rectangle rule"}) \text{ otherwise.} \end{cases} \tag{9.13}$$

5. *Change the matrix entries.* The matrix entries a_{ij} are changed according to the following cases:
$$a_{ij} \leftarrow \begin{cases} 1/d & \text{if } i = i_p \text{ and } j = j_p, \quad (a_{ij} \text{ is the pivot element}) \\ a_{ij}/d & \text{if } i = i_p \text{ and } j \neq j_p, \quad (a_{ij} \text{ is in the pivot row}) \\ -a_{ij}/d & \text{if } i \neq i_p \text{ and } j = j_p, \quad (a_{ij} \text{ is in the pivot column}) \\ a_{ij} - a_{i_p j} a_{i j_p}/d & (\text{"rectangle rule"}) \text{ otherwise.} \end{cases} \tag{9.14}$$

6. *Change the b-values.* The b-values b_i are changed for $i = 0, 1, \ldots, n$ according to the following cases:
$$b_i \leftarrow \begin{cases} b_i/d & \text{if } i = i_p, \quad (b_i \text{ is in the pivot row}) \\ b_i - a_{i j_p} b_{i_p}/d & (\text{"rectangle rule"}) \text{ otherwise.} \end{cases} \tag{9.15}$$

Therefore, as long as they are not in a pivot row or a pivot column, the tableau entries a_{ij}, b_j and c_j are determined by the *"rectangle rule"*

$$w \ \leftarrow \ w - \frac{uv}{d} \qquad \begin{matrix} \boxed{d} & \cdots & u \\ \vdots & & \vdots \\ v & \cdots & w \end{matrix} \tag{9.16}$$

Here $d = a_{i_p j_p}$ is the (old) value of the pivot element, $i \neq i_p$, $j \neq j_p$, and alternatively one of each of the following rows holds:

$$u = \begin{cases} a_{i_p j} \\ b_{i_p} \\ a_{i_p j} \end{cases}, \qquad v = \begin{cases} a_{i j_p} \\ a_{i j_p} \\ c_{j_p} \end{cases}, \qquad w = \begin{cases} a_{ij} \\ b_i \\ c_j \end{cases} \tag{9.17}$$

To summarize, the simplex algorithm consists of the following major parts,

[3] "pivot" literally in German: Angel, Dreh-, Angelpunkt; in the context of the simplex method, however: Pivot

1. find the pivot element;
2. change the pivot variables $y_{i_p} \leftrightarrow x_{j_p}$;
3. replace the pivot element by its reciprocal value $1/d$;
4. multiply the rest of pivot row by the reciprocal pivot value $1/d$;
5. multiply the rest of the pivot column by the negative of the reciprocal pivot value $1/d$;
6. apply the rectangle rule for all other entries.

Let us apply the simplex method to our example 9.1. We start with the following left tableau and perform the first three steps.

	x_1	x_2	
z	2	2.2	0
y_1	1	2	100
y_2	2	1	80
y_3	16	15	960
y_4	20	16	1200

$\Longrightarrow$

	x_1	x_2		
z	2	**2.2**	0	
y_1	1	**2**	100	50 $\leftarrow$
y_2	2	**1**	80	80
y_3	16	**15**	960	64
y_4	20	**16**	1200	75

$\Longrightarrow$

	x_1	x_2	
z	2	**2.2**	0
y_1	**1**	**2**	**100**
y_2	2	**1**	80
y_3	16	**15**	960
y_4	20	**16**	1200

We look for the maximal positive value in the z-row, marked by $\downarrow$. This determines the pivot column. Then the right column (the b-values) are divided by the pivot column; the results are listed right of the tableau. The smallest of them, marked by $\leftarrow$, determines the pivot row. The pivot element is boxed, $a_{i_p j_p} = 2$ with $i_p = 1$ and $j_p = 2$.

Now the entry transformations are executed. The saved value is $d = 2$. Then the first z-row value, $c_1 = 2$, is computed by the rectangle rule (9.16) as $c_1 = 2 - 2.2 \cdot \frac{1}{2} = 0.9$; c_2 is in the pivot column, hence it gets the value $c_2 = \frac{1}{2}$. Similarly, the pivot row is divided by -2. To the rest we apply the rectangle rule. Exchanging the indices $i_p + n = 3$ and $j_p = 2$ yields the left tableau below.

	x_1	y_1	
z	0.9	**–1.1**	-110
x_2	**0.5**	**0.5**	**50**
y_2	1.5	**–0.5**	30
y_3	8.5	**–7.5**	210
y_4	12	**–8**	400

$\Longrightarrow$

	x_1	y_1		
z	**0.9**	-1.1	-110	
x_2	**0.5**	0.5	50	100
y_2	**1.5**	**–0.5**	**30**	20 $\leftarrow$
y_3	**8.5**	-7.5	210	24.7
y_4	**12**	-8	400	33.3

Now the procedure repeats. The pivot column is determined by the only (hence maximal) positive value in the z-row, $c_1 = 0.9$; dividing the right b-values by the pivot column yields the values right of the tableau. By the algorithm we find the following tableau.

	y_2	y_1	
z	**–0.6**	-0.8	-128
x_2	**–0.33**	0.67	40
x_1	**0.67**	**–0.33**	**20**
y_3	**5.67**	-4.67	40
y_4	**–8**	-4	160

(9.18)

All elements c_j of the z-row are now negative. Hence the algorithm stops, the optimum is achieved.

What does this tableau tell us? Its interpretation gives the following results:

- $x_2 = 40$: produce 40 units of product 2;
- $x_1 = 20$: produce 20 units of product 1;
- $y_3 = 40$: 40 kg of raw material R are left over;
- $y_4 = 160$: 160 kg of raw material S are left over;
- $y_2 = 0$: machine A works to capacity;
- $y_1 = 0$: machine B works to capacity;
- $b_0 = -128$: the profit is 128 k€.

9.3 What did we do? or: Why simplex?

To analyze the simplex algorithm, we consider the constraints of example 9.1 in more detail. First, we rewrite the inequalities (9.3) as inequalities with respect to straight lines:

$$\begin{aligned}
x_2 &\leqq 50 - \tfrac{x_1}{2}, && \text{esp.: } x_1 = 0 \Rightarrow x_2 \leqq 50, \quad x_1 = 100 \Rightarrow x_2 \leqq 0 \\
x_2 &\leqq 80 - 2x_1, && \text{esp.: } x_1 = 0 \Rightarrow x_2 \leqq 80, \quad x_1 = 40 \Rightarrow x_2 \leqq 0 \\
x_2 &\leqq 64 - \tfrac{16}{15}\, x_1, && \text{esp.: } x_1 = 0 \Rightarrow x_2 \leqq 64, \quad x_1 = 60 \Rightarrow x_2 \leqq 0 \\
x_2 &\leqq 75 - \tfrac{5}{4} x_1, && \text{esp.: } x_1 = 0 \Rightarrow x_2 \leqq 75, \quad x_1 = 60 \Rightarrow x_2 \leqq 0.
\end{aligned}$$

On the right hand there are given the intersections of the straight lines with the x_1- and x_2-axes. Graphically, the situation is given in figure 9.1. Each constraint is represented by its

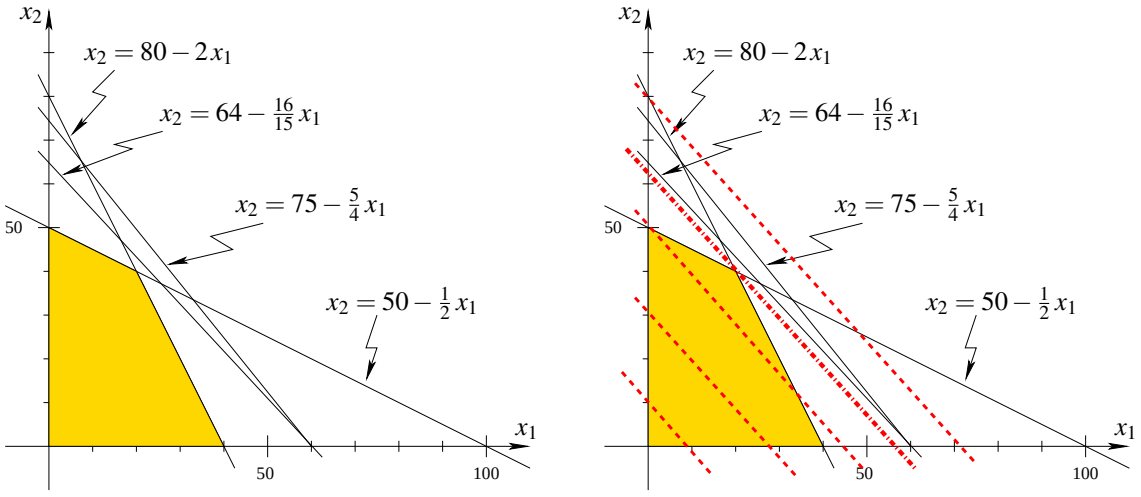


Figure 9.1: Left figure: the constraint lines of example 9.1; the respective inequality "$\leqq$" is geometrically represented by the region below the line, the inequalities $x_1, x_2 \geqq 0$ by the positive quadrant above the x_1-axis and left of the x_2-axis. The shaded region therefore denotes all possible solutions (x_1, x_2). Right figure: the same sketch, added some possible z-lines (dashed); the maximum meeting the shaded region (dashed-dotted line) is the solution.

straight line. All together, they form the shaded region of possible solutions (x_1, x_2). (Note that a solution is a *point* in the diagram!) Also you find various parallel lines for the profit value z ("contour lines"). On each line the profit is equal. The highest line meeting the shaded region is the maximum profit.

9.3.1 What actually is a simplex?

Let $p,n \in \mathbb{N}$, $p \leqq n$. A *p-simplex* in $\mathbb{R}^n$ is a subset of $\mathbb{R}^n$ (a "convex polygon") consisting of $p+1$ points ("vertices") none of which is a sum of the others, if considered as vectors ("linearly independent"). For example, four points in a plane in $\mathbb{R}^3$ are not a simplex, but a tetrahedron is, cf. figure 9.2.

It is now easy to see that the graph of objective function $z = c \cdot x$ forms a plane in $\mathbb{R}^n$ whose intersections with the x_i-axes yield the vertices of a simplex, cf. the dashed lines in figure 9.1. The maximum profit is represented by the z-value plane that intersects the allowed region at its 'outermost' vertex. Geometrically, solving the linear problem (9.8) means shifting the z-value plane, the 'upper side' of the z-value simplex until it reaches this point.

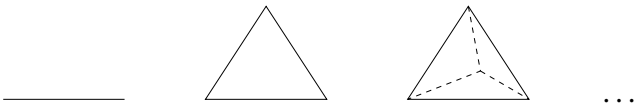


Figure 9.2: A simplex in $\mathbb{R}$, in $\mathbb{R}^2$, and in $\mathbb{R}^3$. A simplex in $\mathbb{R}^n$ has at most $n+1$ vertices.

9.4 Duality

What do we have to do if we want to solve a linear *minimum* problem? The simplex algorithm only can be applied to linear maximum problems, $z \to \max$. A first idea is to consider the modified objective function $z' = -z$, but this leads into a dead-end street: usually the relevant coefficients then are negative and we have no chance to choose a pivot column.

The solution is *duality*. In general, duality is a fascinating and powerful relation between two objects being in different classes (or contexts) but having the same or equivalent properties. For example, in every-day life the mirror reflection is a duality, since the mirror image of a geometrical object can be uniquely mapped to its original ... and vice versa, by the same operation. Another, more subtle examle is the geometric duality of points and straight lines in a

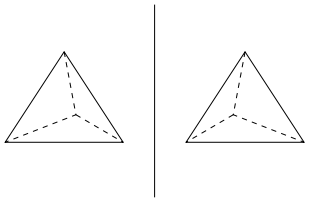


Figure 9.3: Mirror symmetry is a duality.

plane. The statement "Two points determine a straight line" has as dual statement "Two straight lines determine a point."

Once a duality is known, a given problem can be transformed by it to another problem, the "dual problem," which sometimes is much easier to solve.It is a remarkable property that *minimum problems are dual to maximum problems.* That means that we can

1. transform the minimum problem to a maximum problem,
2. solve it by the simplex algorithm, and
3. transform the solution back by reading the non-vanishing y-values from the corresponding z-row entries.

However, what is the duality operation? The transformation rule of forming the dual problem (the "mirror image") of a minimum problem is given by the following table.

minimum problem	**dual problem**
$z^*(\vec{y}) = \vec{b}^* \cdot \vec{y} \ \rightarrow \ \min$	$z(\vec{x}) = \vec{c}^* \cdot \vec{x} \ \rightarrow \ \max$
constraints:	constraints:
$A^* \cdot \vec{y} \geqq \vec{c}$	$A \cdot \vec{x} \leqq \vec{b}$
variables:	variables:
$\vec{y} \geqq 0$	$\vec{x} \geqq 0$

(9.19)

Here A^* denotes the *conjugate*[4] (that is, since we only deal with real matrices, simply the transpose[5]) of A, i.e. the matrix resulting from interchanging its rows and columns. In particular, the following correspondences between the variables and the dual slack variables can be shown [10].

minimum problem	**dual problem**
variable y_i	slack variable y_i of the i-th constraint
slack variable x_j of the j-th constraint	variable x_j

(9.20)

Example 9.2. We first consider a trivial minimum problem to clarify the principles. Assume a company has to buy 5 qu of a product whose production costs are limited by the condition that three qu of it cannot be got for less than 12 €. What is the price y per qu of the product to minimize the total costs? (It is immediately clear that the solution is $y = 4$, but let us see how the solution startegy works!)
Mathematical formulation: The objective function expressing the total costs is $z(y) = 5y$, and the constraint is given by $3y \geqq 12$. In matrix notation we thus achieve

$$z^*(y) = by \ \rightarrow \ \min, \qquad A^* y \geqq c, \tag{9.21}$$

with the one-dimensional vectors y ("$= (y)$"), $b = 5$, $c = 12$. and the (1×1)-matrix $A^* = 3$. Since $A = A^* = 3$, we achieve the dual problem

$$z(x) = 12x \ \rightarrow \ \max, \qquad 3x \leqq 5. \tag{9.22}$$

It can be solved by the simplex algorithm:

	x	
z	12	0
y	$\boxed{3}$	5

$\Longrightarrow$

	y	
z	-4	-20
x	$\frac{1}{3}$	$\frac{5}{3}$

The maximum problem thus is solved for $x = \frac{5}{3}$, and the original minimum problem for $y = 4$, yielding total costs of $z(3) = 5 \cdot 4 = 20$. □

9.4.1 Economical interpretation of duality in linear optimization problems

In example 9.2, the objective function

$$z(x,y) = xy \qquad (x\text{: quantity, } y\text{: price per quantity}) \tag{9.23}$$

[4] in German: "die Konjugierte"
[5] in German: "die Transponierte"

is optimized with respect to two different point of views: the minimum problem (9.21) searches to minimize the production costs for a given quantity ($x = 5$ qu) and a lower price limit ($3y \geqq 12$ €); the dual maximum problem (9.22) aims to maximize the quantity x for a given price ($y = 12$ €/qu) and an upper quantity limit ($3x \leqq 5$ qu).

Therefore, the minimum problem (9.21) is related to the *demand-sided* viewpoint (the producer pays for ressources, i.e. has costs), whereas the dual problem (9.22) is viewed by the *supplier* (who gains the production prices). *Duality in economical optimization problems often reflects the different points of view of a demander and a supplier.*

Example 9.3. Let be given the minimum problem

$$z^*(\vec{y}) = 100y_1 + 80y_2 + 960y_3 + 1200y_4 \longrightarrow \min \tag{9.24}$$

under the constraints

$$\begin{array}{rcrcrcrcr} y_1 & + & 2y_2 & + & 16y_3 & + & 20y_4 & \geqq & 2 \\ 2y_1 & + & y_2 & + & 15y_3 & + & 16y_4 & \geqq & 2.2 \end{array} \tag{9.25}$$

In other words, we have $z^*(\vec{y}) = \vec{b}^* \cdot \vec{y}$ under the constraints $A^*\vec{y} \geqq \vec{c}$, where

$$\vec{b}^* = (100, 80, 960, 1200), \quad A^* = \begin{pmatrix} 1 & 2 & 16 & 20 \\ 2 & 1 & 15 & 16 \end{pmatrix}, \quad \vec{c} = \begin{pmatrix} 2 \\ 2.2 \end{pmatrix}.$$

Thus we see immediately that the dual problem is exactly example 9.1 (p. 87). Solving it by the simplex algorithm yields the final tableau (9.18), from which we can read the solution for $\vec{y}$ in the z-row:

$$y_1 = 0.8, \quad y_2 = 0.6, \quad y_3 = y_4 = 0, \qquad \text{or} \quad \vec{y} = (0.8, 0.6, 0, 0)^*.$$

□

Example 9.4. A firm produces a product from two raw materials, where for each 2 kg of the first material there is always at least 1 kg of the second material. It should be bought at least 50 kg of the first material, and at least 100 kg of both in total. The price of the first material is 6 €/kg, whereas the price of the second material is 9 €/kg. What quantities of both materials must be bought such that the costs are minimal?

Solution. First we formulate the problem mathematically. Denote the quantity in kg of the first material by y_1, and the quantitiy of material 2 by y_2. The first condition mentioned means that we always have at most twice as much of material 1 than of material 2, i.e., $y_1 \leqq 2y_2$. This can be rewritten to a restriction in the form: $2y_2 - y_1 \geqq 0$. Together with the two other mentioned restrictions we thus have the systems of inequalities

$$\begin{array}{rcrcr} -y_1 & + 2y_2 & \geqq & 0, \\ y_1 & & \geqq & 50, \\ y_1 & + y_2 & \geqq & 100. \end{array}$$

The objective function reads

$$z^*(y_1, y_2) = 6y_1 + 9y_2 \quad \longrightarrow \quad \min.$$

In matrix notation, this is $z^*(\vec{y}) = \vec{b}^*\vec{y} \to \min$, under the constraints $A^*\vec{y} \geqq \vec{c}$, where

$$A^* = \begin{pmatrix} -1 & 2 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} 0 \\ 50 \\ 100 \end{pmatrix}, \qquad \vec{c} = \begin{pmatrix} 6 \\ 9 \end{pmatrix},$$

This is a minimum problem. Its dual is $z(\vec{x}) = \vec{c}^* \vec{x} \to \max$, under the constraints $A^* \vec{x} \leqq \vec{c}$. This gives the following simplex tableau.

	x_1	x_2	x_3	
z	50	100	0	0
y_1	1	$\boxed{1}$	-1	6
y_2	0	1	2	9

$\Longrightarrow$

	x_1	y_1	x_3	
z	50	-100	100	-600
x_2	1	1	-1	6
y_2	-1	-1	$\boxed{3}$	3

$\Longrightarrow$

	x_1	y_1	y_2	
z	$-\frac{50}{3}$	$-\frac{200}{3}$	$-\frac{100}{3}$	-700
x_2	1	1	$\frac{1}{3}$	7
x_3	$-\frac{1}{3}$	$-\frac{1}{3}$	1	1

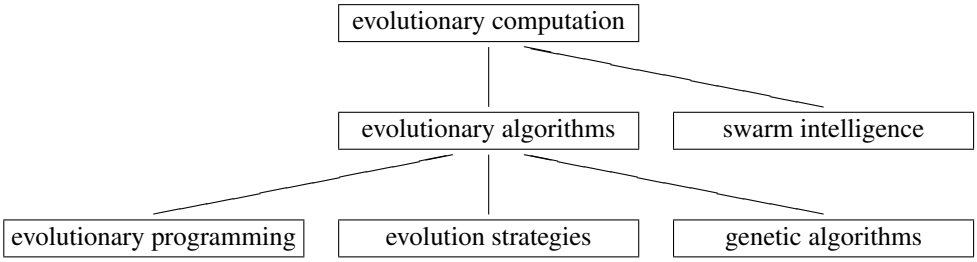
Therefore, $200/3 = 66.\overline{6}\,\text{kg}$ of the first raw material and $100/3 = 33.\overline{3}\,\text{kg}$ of the second raw material should be bought. This yields total buying costs of 700 €. □

Chapter 10

Genetic algorithms

10.1 Evolutionary algorithms

There is some confusion with respect to the terminology of genetic algorithms and related areas. This is mainly due to the fact that different ideas in the field developed independently from each other at different times. The idea of applying evolutionary principles to computer sciences first emerged in the 1960's in the context of finite automata. For this approach the term *evolutionary programming* was established. Also in the 1960's and in a different context, experimental optimization was summarized to the notion *evolution strategies*, done by Schwefel and Rechenberg in Berlin.[1] In the 1970's, genetic algorithms was introduced by Holland and Fogel in the USA. Nowadays, all these terms are comprised to the branch *evolutionary algorithms*. Whereas in



evolutionary programming the parameters of the program executing the optimization are varied, evolution strategies and genetic algorithm differ only in that the search space of evolution strategies is real hyperspace and in that the mutation is adapted during the evolution process. For details and further historical remarks see [15, 43].

10.2 Basic notions

There is a large class of interesting problems for which no reasonably fast algorithm could be developed so far. Many of these problems are optimization problems. Given such a "hard" optimization problem it is sometimes possible to find an efficient algorithm whose solution is *approximately* optimal. For some hard optimization problems we can use probabilistic algorithms as well — these algorithms do not guarantee the optimum value, but by randomly choosing sufficiently many "witnesses" the probability of error may be made as small as wished. Examples of such algorithms are "Monte Carlo techniques" and especially "simulated annealing." A good introduction to these topics and to the subject of genetic algorithms in general is given in [15].

In general, any abstract task to be accomplished can be thought of as solving a problem which, in turn, can be perceived as a search through a space of potential solutions. Since we

[1] *http://geneticargonaut.blogspot.com/2006/03/evolutionary-computation-classics-vol.html* (visited on Feb 26, 2007)

are searching for "a best" solution, we can view this task as an optimization process. For small spaces, classical exhaustive, or brute-force, methods usually suffice, for larger spaces special techniques from the area of *artificial intelligence* must be employed.

Genetic algorithms are among such techniques. They are stochastic algorithms whose search methods model the natural phenomenon of *evolution*: genetic inheritance, mutation, and selection. In evolution, the problem each species faces is one of searching for beneficial adaptions to a complicated and changing environment. The "knowledge" that each species has gained is embodied in the makeup of the chromosomes of its members.

Example 10.1. (The rabbit example) [31]. At any given time there is a population of rabbits, some of which are smarter and faster than the others. The faster and smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits. Of course, some of the slower and dumber rabbits will survive just because they are lucky.

This surviving population of rabbits starts breeding. The breeding results in a good mixture of rabbit genetic material: some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits, and so on. And on the top of that, Nature throws in a "wild hare" every now and then by mutating some of the rabbit genetic material.

What is the ultimate effect? The resulting baby rabbits will, on average, be faster and smarter than those in the original population because more faster and smarter parents survived the foxes.[2] □

A genetic algorithm follows a step-by-step procedure that closely matches the story of the rabbits. Genetic algorithms use a vocabulary borrowed from natural genetics:

- We talk about *individuals* or *genotypes* in a *population*
- An individual is determined by its *chromosomes*. Each cell of an organism of a given species carries a certain number of chromosomes (a human being, e.g., has 46 of them); however, we talk about one-chromosome individuals only ("haploid chromosomes"). Often you will thus find the notions "chromosome," "individual," and "genotype" being used as synonyms.[3]
- Chromosomes are made of units, the *genes*. They are arranged in a linear succession. Genes are located at certain places of the chromosome, called *loci*.
- Any feature of individuals (such as hair color) can manifest itself differently; the gene is said to be in several states called *alleles*, i.e., values.

Table 10.1 shortly compares the original biological meaning of different notions and their meaning with respect to genetic algorithms.

An evolution process running on a population of chromosomes corresponds to a search through a search space S of feasible solutions. Such a search requires balancing two apparently conflicting objectives: exploiting the best solutions and exploring the solution space sufficiently. Random search is a typical example of a strategy which explores the solution space *ignoring* the exploitations of promising regions of the space. Genetic algorithms are a class of general purpose ("domain independent") search methods which strike a remarkable balance between the conflicting strategies of exploring and exploiting.

[2] Of course the foxes undergo a similar process, otherwise the rabbits might become too fast and smart for them.

[3] Strictly speaking, in biology there is the hierarchy chromosome → genotype → phenotype = individual. Especially, there may be several genotypes resulting in the same phenotype, mainly because a phenotype is influenced by the environment. Usually, these distinctions are not adhered to in evolutionary algorithms.

Notion	Biology	Genetic Algorithms
chromosome	blueprint of the individual organism, package for carrying DNA	number array or string, specifying the data structure of solutions in search space S
gene	inheritance unit consisting of DNA, occupying a segment in a chromosome and determining a characteristic of the organism ("a gene for the eye color")	region of the chromosome
allele	form of the gene, specifying a characteristic of the organism ("eye color green")	value of the gene
locus	place of a gene in the chromosome	position of the gene
phenotype	outward appearance of the organism	effective solution
genotype	structure of the chromosome	formal coding of a chromosome; different genotypes may result in the same phenotype
generation	cohort, set of organisms born at the same time	iterative set of chromosomes
fitness	capability of an individual of certain a genotype to reproduce	evaluated quality of the solution represented by the chromosome, capability of the individual

Table 10.1: Important genetic notions

10.3 The "canonical" genetic algorithm

A genetic algorithm for a paricular problem must specify the following components.

- The *genetic representation* $S = \{x\}$, for a feasible solution x to the problem: x are the individuals or chromosomes, and the set S is their data structure. Usually, in a genetic algorithm the chromosome is a binary string of length n,

  Thus usually, $S \subseteq \{0,1\}^n \subset \mathbb{Z}^n$, i.e., genetic algorithms apply to combinatorial optimization problems. Usually, each single bit is a gene.

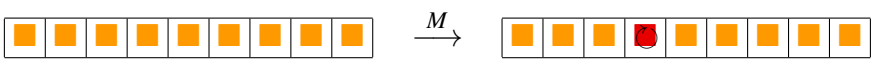
- An objective function $f : S \to \mathbb{R}$, called *fitness function* in case of evolutionary algorithms, which plays the role of the environment, the "selection." Therefore, $f(x)$ is the fitness of the feasible solution $x \in S$.

- *Genetic operators* which generate and alter the chromosomes of the children ("alteration"). There are two types of genetic operators:

  - A *crossover operator* or *recombination operator*, $CX : S^k \to S$ which combines genes of two ($S^2 = S \times S$) or more ($S^k = \underbrace{S \times \cdots \times S}_{k-\text{times}}$) individuals, called the *parents*:

    Often, a crossover operator simultaneously creates two children from two parents, with the roles of the two parents exchanged. There may be several crossover operators acting in a genetic algorithm.

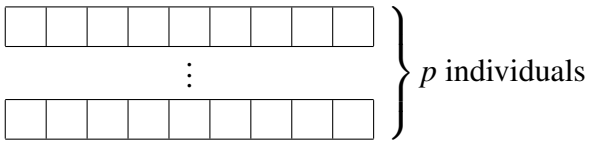
  - A *mutation operator* $M : S \to S$ which changes one or several genes randomly,



- Various parameter values used by the genetic algorithm, such as population size, probabilities of applying the genetic operators, etc.

Moreover, a design of a genetic algorithm has to specify the following subroutines.

- *Initialization.* This subroutine creates an *initial population* $P(0)$ of feasible solutions. Usually, the population size p remains constant over the execution of the algorithm,



  (Note that an individual here is identified with its chromosome.) Usually, the initial population is given by creating p solutions by random.

- *Selection. ("Survival of the fittest")* This subroutine selects the parents from the current population to reproduce the next generation. There are three popular selection principles, *truncation selection* where a fixed percentage of the best individuals are chosen, *roulette-wheel selection* where individuals are selected with a probability proportional to their fitness, and *tournament selection* where a pool of individuals is chosen at random and then the better individuals are selected with predefined probabilities. Most selection are stochastic, such as the latter two, so that a small proportion of less fit solutions are selected. This helps keep the diversity of the population large, preventing premature convergence on local but non-global optima.

- *Reproduction.* This subroutine generates the next generation from the selected parents of the current population. Besides the application of the genetic operators to the parents, the routine must define to what extend the parent generation and the children survive. A genetic algorithm with a reproduction scheme in which the best individual of a generation is guaranteed to survive is said to obey the *elite principle* [15, §5.2].

- *Termination.* A terminating condition has to be specified. In difference to a deterministic optimization algorithm, a genetic algorithm running a finite time can never yield a global optimum with certainty. So there has to be implemented a highhanded terminating condition, e.g., if the number of generations has reached a fixed limit, or if the highest ranking solution's fitness has reached a plateau such that successive iterations no longer produce better results.

A genetic algorithm therefore is a probabilistic algorithm which maintains a population of p individuals,

$$P(t) = (x_1(t), \ldots, x_p(t)) \in S^p$$

in each iteration step t, called the t-th *generation*. For each individual $x_j(t)$ of this generation, its fitness $f(x_j(t))$ is evaluated. Then a new population, the next generation $t+1$, is formed by selecting the fitter individuals ("selection step"). Some members of the new population undergo transformations by means of the genetic operators to form new solutions ("reproduction step"). After some number of generations the program generates better and better individuals, hopefully the best individual represents a near-optimum solution. To summarize, the "canonical" genetic

algorithm reads in pseudocode:

```
algorithm genetic {
   t ← 0;
   initialize P(t);
   evaluate P(t);
   while ( !terminating condition ) {
      t++;
      select P(t) from P(t−1);
      reproduce P(t);
      evaluate P(t);
   }
}
```
(10.1)

10.4 The 0-1 knapsack problem

A wanderer has a knapsack with a limited capacity of total weight. He has to select among a finite set of objects each of which has a certain value and a certain weight. Which of the items should he pick in his knapsack to maximize the total value, without violating the maximum weight?

Example 10.2. Assume that the wanderer has a knapsack with a maximum load of 5 kg and wants to select the objects given by the following table.

Object	**Weight**	**Value**
A	2 kg	8 €
B	3 kg	10 €
C	1 kg	3 €
Maximum load	5 kg	

Which objects should he put into the knapsack? □

To construct a genetic algorithm for this problem, we first recognize that it is an optimization problem, namely a maximum problem. What is its search space, what is its objective function?

- Given n objects, the search space S is most naturally given by a binary string $x \in \{0,1\}^n$ of length n, where the k-th bit indicates whether the k-th object is picked into the knapsack: $x_k = 0$ means that it is not, $x_k = 1$ means that it is. For instance, Example 10.2 implies a search space which consists of vectors (x_1, x_2, x_3) with x_1, x_2, $x_3 \in \{0,1\}$, and the candidate solution $x = (0,1,0)$ means that only object B is put into the knapsack. However, we have the constraint that the maximum load of the knapsack must not be exceeded: this is most easily expressed by a weight vector $w = (w_1, \ldots, w_n)$ where w_k denotes the weight of object k, and thus the weight constraint reads $\sum_k^n w_k x_k \leqq w_{\max}$. Therefore, the search space is determined by

$$S = \left\{ x \in \{0,1\}^n : \sum_{k=1}^{n} w_k x_k \leqq w_{\max} \right\} \tag{10.2}$$

  where $x = (x_1, \ldots, x_n)$. Formally, we can take the n components x_k and w_k as (column) vectors x and w, and the constraint may be written as $w^* \cdot x \leqq w_{\max}$.

- The objective function for the knapsack problem is obvious, it is the total value of the load. Defined on the search space, we therefore have $f : S \to \mathbb{R}^+$,

$$f(x) = \sum_{k=1}^{n} v_k x_k \tag{10.3}$$

where $v = (v_1, \ldots, v_n)$ is the "value vector" of the n objects, v_k denoting the value of the k-th object. In Example 10.2 we have $v = (8, 10, 3)$, and the candidate solution $x = (0, 1, 0)$ has a fitness of $f(x) = 10$. With it, each generation can be evaluated.

- Creating the initial population by random, we have to tackle the problem that a random binary string $x \in \{0, 1\}$ is not necessarily a feasible solution, i.e., it may be $x \notin S$, because it exceeds the maximum load. There are two strategies for this case, first to repair the chromosome such that the corresponding solution obeys the constraint, or second to give it a "penalty fitness," say a vanishing value.

10.5 Difficulties of genetic algorithms

10.5.1 Premature convergence

For practical applications of genetic algorithms, the question is essential: How fast does the algorithm converge to a global optimum? So far, little is theoretically known about the convergence of a general genetic algorithm. There are too many degrees of freedom in the effects of the various genetic and selection operators. What is known is that the "canonical" genetic algorithm above with elite principle ("the fittest individual(s) of each generation survives certainly") does converge, but that the user should have some time to wait, because the optimum is reached for $t \to \infty$.

Thus, as for any probabilistic algorithm, breaking up a genetic algorithm after finite time, one can never be sure to have reached a global optimum. Even if there is not made any improvement for some number of generations, it is a good guess that the algorithm has found a local optimum, i.e., a suboptimum. This behavior is called *premature convergence*. It can happen in particular if the region of attraction of a global optimum is small as compared to region of attractions of suboptima.

It is the selection operator which has main responsibility for premature convergence. The roulette-wheel selection empirically leads to a smaller selection pressure [15, §5.3]. In this way, also chromosomes which initially are less fit get the chance to evolve further. So, this is the general problem of any genetic algorithm: to adjust the parameters and the operators in such a way that a optimal balance between *exploration* and *exploitation* is found, that is, between wide scanning of the search space and simultaneous support to further develop good individuals in the population.

10.5.2 Coding

Most genetic algorithms act on search spaces containing binary strings as chromosomes, i.e., $S \subset \{0, 1\}^n$. Usually, such a string is the binary representation of an integer which expresses parameter values of the optimization. However, the binary code has the property that successive numbers may have a large *Hamming distance*. The Hamming distance d_H between two binary strings is defined by the number of positions in which they differ. For example,

$$d_H(01111, 10000) = 5. \tag{10.4}$$

Often the fitness depends on the numerical value of a chromosome, the mutation of single bits in a chromosome whose fitness is close to the optimum may lead to negative effects, cf. Table 10.2. For instance, consider a population of 3-bit strings x and a fitness function $f(x) = 8x - x^2$. Then the maximum is achieved for $x = 100_2$. If we had a population $P = \{0, 3, 5\}$ (in decimal notation) then by $f(3) = f(5) = 15$ and $f(0) = 0$ we would have the curious situation that although the chromosomes $x = 011$ and $x = 101$ have the same good fitness, a mutation of

Decimal number	Binary code		Gray code	
	Code	d_H	Code	d_H
0	000	1	000	1
1	001	2	001	1
2	010	1	011	1
3	011	3	010	1
4	100	1	110	1
5	101	2	111	1
6	110	1	101	1
7	111		100	

Table 10.2: The numbers 0, ..., 7 represented in standard ("natural") binary code and in Gray code. The Hamming distance d_H between two successive numbers is always 1 for the Gray code, whereas it varies for the binary code.

a single bit could possibly change 101 to the optimum 100, but 011 has to change *all three bits*. Even the much worser solution $x = 000$ only needs to change a single bit to become the optimum.

A commonplace solution to this problem is the use of a *Gray code*. It is a one-to-one mapping γ from the natural binary code and is defined as $g = \gamma(b)$, where $g = g_{n-1} \dots g_1 g_0$ is the Gray code string, $b = b_{n-1} \dots b_1 b_0$ is the binary code string, and

$$g_i = b_{i+1} \oplus b_i \qquad (i = 0, \dots, n-1) \tag{10.5}$$

with $b_n = 0$. Here $\oplus$ denotes the bitwise XOR operation (in Java: ^). Thus the Gray code is calculated from the binary code by the following scheme:

$$\begin{array}{cccccc} & b_{n-1} & b_{n-2} & \cdots & b_1 & b_0 \\ \oplus & 0 & b_{n-1} & \cdots & b_2 & b_1 \\ \hline = & g_{n-1} & g_{n-2} & \cdots & g_1 & g_0 \end{array} \tag{10.6}$$

With logical bit operators, this is simply expressed as

```
g = b^(b » 1)
```

In Java, a method converting a `long` integer into a Gray code string may therefore be implemented as follows:

```
public static String grayCode(long b) {
   return Long.toBinaryString( b^(b >> 1) );
}
```

The inverse mapping is given recursively by

$$b_{n-1} = g_{n-1}, \qquad b_i = b_{i+1} \oplus g_i \qquad (i = n-2, \dots, 0). \tag{10.7}$$

Accordingly, the inverse method converting a Gray code into a `long` integer could read as follows.

```
public static long toLong(String grayCode) {
   long b = Long.parseLong(grayCode,2), g = 0;
   for(int i = grayCode.length() - 1; i >= 0; i--) {
      g += ( (g & (1 << i+1) ) >> 1) ^ (b & (1 << i));
   }
   return y;
}
```

or alternatively, if you prefer array representations to logical bitwise operators,

```
public static long toLong(String grayCode) {
   char[] g = grayCode.toCharArray();
   char[] b = new char[gray.length];
   b[0] = g[0];   // note that in an array the highest bit has index 0!
   for (int i = 1; i < g.length; i++) {
      b[i] = (b[i-1] == g[i]) ? '0' : '1';  // i.e.: b[i] = b[i-1] XOR g[i]
   }
   return Long.parseLong( String.valueOf(binary), 2);
}
```

10.6 The traveling salesman problem

In the 1990's there have been several attempts to approximate the TSP (Example 6.2) by genetic algorithms; here one of them is presented.

First we note that a binary string is not an appropriate chromosome representation. In a binary representation of an n city TSP, each city could be coded as a string of $\lfloor \log_2 n \rfloor + 1$ bits; thus a chromosome as a complete tour is a string of $n(\lfloor \log_2 n \rfloor + 1)$ bits. A mutation now can result in sequence of cities which is not a tour: we can get the same city twice in a sequence. Moreover, for a TSP with 20 cities, where we need 5 bits to represent a city, some 5-bit sequences (10101, e.g.) do not correspond to a city. Similar considerations are present when applying crossover operators. Clearly, if we use mutation and crossover operators as random operators, we would need some sort of "repair operator" which would move a chromosome back into the solution space S.

But there is a better representation, the integer vector representation. Instead of using repair operators, we can incorporate the knowledge of the problem into the representation. This "intelligently" avoids building an illegal individual. Let the vector

$$\vec{v} = (i_1, i_2, \ldots, i_n) \qquad \text{with } i_j \in \{1, 2, \ldots, n\} \quad (j = 1, \ldots, n) \tag{10.8}$$

represent the tour from i_1 to i_2, from i_2 to i_3, ..., form i_{n-1} to i_n, and from i_n back to i_1,

$$i_1 \to i_2 \to \ldots \to i_{n-1} \to i_n \to i_1.$$

($\vec{v}$ is a so-called "permutation"). We can initialize the population by a random sample of $\vec{v}$. (We can alternatively use a heuristic algorithm for the initialization process to get "preprocessed" outputs.)

The evaluation of a chromosome is straightforward. Given the cost of travel c_{ij} between cities i and j, we can easily calculate the total cost of the entire tour with the fitness function $f : {\mathbb{Z}_n}^n \to \mathbb{R}$,

$$f(\vec{v}) = \sum_{j=1}^{n-1} c_{i_j, i_{j+1}} + c_{i_n, i_1}. \tag{10.9}$$

(cf. equation (8.11), p. 85). In the TSP we thus search for the best ordering of cities in a tour. It is relatively easy to come up with (unary) mutation operators. However, there is little hope of finding good orderings (not to mention the best ones), because a good ordering needs not to be located "near" another good one. The strength of genetic algorithms especially arises from the structured information exchange of crossover combinations of highly fit individuals. So, what we need is a crossover operator that exploits important similarities between chromosomes. For that purpose we need an OX operator. Given two parents $\vec{v}$ and $\vec{w}$, OX builds offspring $\vec{u}$ by

choosing a subsequence of a tour from one parent and preserves the relative order of cities (without those of the subsequence) from the other parent. For example,

$$\vec{v} = (1,2,3,4,5,6,7,8,9,10,11,12), \quad \vec{w} = (7,3,1,11,4,12,5,2,10,9,6,8). \tag{10.10}$$

If the chosen part from parent $\vec{v}$ is $(4,5,6,7)$, we have to cross out the cities in $\vec{w}$, i.e. $\vec{w}' = (3, 1, 11, 12, 2, 10, 9, 8)$, start at city 1, and insert the chosen subsequence at the same position as in parent $\vec{v}$. This gives the child

$$\vec{u} = (1,11,12,4,5,6,7,2,10,9,8,3). \tag{10.11}$$

As required from a genetic algorithm, the child bears a structural relationship to both parents. The roles of the parents $\vec{v}$ and $\vec{w}$ can then be reversed to construct a second child.

A genetic algorithm based on the above operator outperforms random search, but leaves much room for improvements. Typical results from the algorithm (average over 20 random runs) as applied to 100 randomly generated cities gave, after 20 000 generations, a value of the whole tour 9.4% above the optimum.

10.7 Axelrod's genetic algorithm for the prisoner's dilemma

The prisoner's dilemma

Two suspects, A and B, are arrested by the police. held in separate cells, unable to communicate with each other. The police have insufficient evidence for a conviction and visit each of them to offer the same deal: if one testifies for the prosecution against the other and the other remains silent, the betrayer goes free and the silent accomplice receives the full 10-year sentence. If both stay silent, the police can sentence both prisoners to only $\frac{1}{2}$ years in jail for a minor charge. If each betrays the other, each will receive a two-year sentence. Each prisoner must make the choice of whether to betray the other ("D" for "defect") or to remain silent ("C" for "cooperate"). However, neither prisoner knows for sure what choice the other prisoner will make. So the question this dilemma poses is: What will happen? How will the prisoners act?

(zur anderen Seite)

The prisoner's dilemma can be played as a game between two players, where at each turn, each player either defects or cooperates with the other prisoner. The players then score according to the payoffs listed in table 10.3. In game theory [14, 25], one often uses a strategy

Player A	Player B	P_A	P_B	Commentary
defect	defect	-2	-2	punishment for mutual defection
defect	cooperate	0	-10	temptation to defect and betrayer's payoff
cooperate	defect	-10	0	betrayer's payoff and temptation to defect
cooperate	cooperate	$-\frac{1}{2}$	$-\frac{1}{2}$	reward for mutual cooperation

Table 10.3: Payoff table for prisoner's dilemma game; P_j is the payoff for player $j \in \{A,B\}$.

table, where the left head column are player A's strategies, and the head row contains player B's strategies. In each table cell, the first entry is player A's payoff for the corresponding strategy profile, the second entry is player B's, see table 10.4. (C means cooperate, D means defect.)

The "C–C" strategy in a multi-move prisoner's dilemma game is a so-called "Nash equilibrium." That means that each player's strategy is an optimal response to the other players' strategies.

Example 10.3. *(Multi-move prisoner's dilemma in economics)* Consider two oligopolists A and B competing on a single market. In each seasonal period, each of them has the choice to

	C	D
C	$(-\frac{1}{2};-\frac{1}{2})$	$(-10;0)$
D	$(0;-10)$	$(-2;-2)$

Table 10.4: Strategy table for prisoner's dilemma game. The first entry a in each box (a,b) is player A's payoff for the corresponding strategy profile; the second one b is player B's payoff.

	C	D
C	(3; 3)	$(-2; 5)$
D	(5;−2)	$(-1;-1)$

Table 10.5: Strategy table for a single move in the economic prisoner's dilemma game, where C implies the decision to make a fair price for a product, and D refers to dumping the product.

"cooperate" (C) and to make a fair price for a given product, or to "dump" (D) the product and make a dirt-cheap price for it. The expected payoff (in Mio €) for one firm depends on the simultaneous decision of the competitor according to Table 10.5. How should each firm decide to make the price? □

A *strategy* in game theory is a plan of unique moves to be made after each possible past constellation of moves; such a constellation can depend on only the last move of the rival, but also on a series of past moves. In other words, a strategy is a collection of precise answers to all possible questions.

We will now consider how a genetic algorithm might be used to learn a strategy for the prisoner's dilemma. We have to maintain a population of "players", each of whom has a particular strategy. Initially, each player's strategy is chosen at random. Thereafter, at each step, players play games and their scores are noted. Some of the players are then selected for the next generation, and some of those are chosen to mate. When two players mate, the new player created has a strategy constructed from the strategies of its parents (crossover). A mutation, as usual, introduces some variability into players' strategies by random changes on representations of these strategies.

Chromosome represention of a strategy

First of all, we need some way to represent a strategy, i.e. a possible solution. For simplicity, we will consider strategies that are deterministic and use the outcomes of the three previous moves to make a choice in the current move. Since there are 4 possible outcomes of each move, there are $4^3 = 64$ different histories of the three previous moves, plus the decision.

A strategy of this type can be specified by indicating what move is to be made for each of these possible histories. Thus, a strategy can be represented by a string of 6 bits (or D's and C's), indicating what move is to be made for each of the $64 = 2^6$ possible histories, plus the 1 bit for the decision. making a total of 7 bits for a chromosome, i.e.,

$$x = \underbrace{\boxed{(a_{-3}, b_{-3})}\boxed{(a_{-2}, b_{-2})}\boxed{(a_{-1}, b_{-1})}}_{\text{3 previous moves}}\underbrace{\boxed{a_0}}_{\text{next move}}$$

with $a_j, b_j \in \{\text{C, D}\}$, for $i, j = -3, \ldots, 0$; the a's are this player's move (C for cooperate, D for defect) and the b's are the other player's moves.

Outline of Axelrod's genetic algorithm

Axelrod's genetic algorithm to learn a strategy for the prisoner's dilemma works in four stages, as follows.

1. *Choose an initial population.* Each player is assigned a random string of 71 bits, representing a strategy.
2. *Test each player to determine its fitness.* Each player uses the strategy defined by its chromosome. The player's score is its average over all games it plays.
3. *Select players to breed.* A player with an average score is given one mating; a player scoring one standard deviation above the average is given two matings; a player scoring one deviation below the average is given no matings.
4. *Pair the fittest.* The successful players are randomly paired off to produce two offspring per mating. Each offspring's strategy is determined from the strategies of its parents, done by using two genetic operations, crossover and mutation.

After these four stages, we get a new population. It will show patterns of behavior that are more like those of the successful individuals of the previous generation.

Experimental results

Running this program, Axelrod obtained quite remarkable results. From a random start, the genetic algorithm evolved populations whose median member was as successful as the best known heuristic algorithm. Some behaviorial patterns evolved in the vast majority of the individuals:

strategy	history	next move
"Don't rock the boat"	(CC)(CC)(CC)	C
"Be provokable"	(CC)CC)CD)	D
"Accept an apology"	(CD)(DC)(CC)	C
"Forget"	(DC)(CC)(CC)	C
"Accept a rut"	(DD)(DD)(DD)	D

10.8 Conclusions

The examples of genetic algorithms in this chapter show their wide applicability. At the same time we observed first signs of difficulties. The representation issues of the traveling salesman problem were not obvious, and the new operator (OX crossover) was far from trivial. What kind of representation difficulties may exist for other problems? On the other hand, how should we proceed in a case where the fitness function is not well defined?[4]

[4]For example, the famous Boolean Satisfiability Problem (SAT) seems to have a natural string representation (the i-th bit represents the truth value of the i-th Boolean variable), however, the fitness function is far from being obvious.

Appendix

Appendix A

Mathematics

A.1 Exponential and logarithm functions

$$\boxed{a^{x+y} = a^x \cdot a^y.} \qquad (a > 0,\ x, y \in \mathbb{R}) \tag{A.1}$$

Let ln denote the natural logarithm, i.e. $\ln x = \log_{\mathrm{e}} x$. The logarithm is the inverse of the exponential function,

$$\mathrm{e}^{\ln x} = x. \qquad \ln \mathrm{e}^y = y. \qquad (x > 0,\ y \in \mathbb{R}) \tag{A.2}$$

Moreover,

$$\boxed{\log_a(xy) = \log_a x + \log_a y,} \qquad \boxed{c \cdot \log_a x = \log_a x^c.} \qquad (x, y > 0,\ c \in \mathbb{R}) \tag{A.3}$$

A change of the logarithm base can be applied according to the rule

$$\boxed{\log_a x = \frac{\log_b x}{\log_b a}.} \qquad (a, b \in \mathbb{N},\ x > 0) \tag{A.4}$$

In particular, $\log_a x = \frac{1}{\ln a} \ln x$.

A.2 Number theory

Integers play a fundamental role in mathematics as well as in algorithmics and computer science. So we will start with the basic notation for them. As usual, $\mathbb{N} = \{1, 2, 3, \ldots\}$ is the set of *natural numbers* or *positive integers*, and

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, \ldots\}$$

is the set of *integers*. The rational numbers $q = m/n$ for $m, n \in \mathbb{Z}$ are denoted by $\mathbb{Q}$. The "holes" that are still left in $\mathbb{Q}$ (note that prominent numbers like $\sqrt{2}$ or π are not in $\mathbb{Q}$!) are only filled by the *real numbers* denoted by $\mathbb{R}$. Thus we have the proper inclusion chain $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$. ("Proper inclusion" means that there are always numbers that are in a set but not in its subset. Do you find an example for each subset-set pair?)

A very important topic in mathematics, especially for the growing area of cryptology, is number theory. We will list here some fundamentals. In this chapter lower case italic letters (such as m, n, p, q, ...) denote integers.

Definition A.1. We say that m *divides* n, in symbols $m \mid n$, if there is an integer k such that $n = km$. We then call m a *divisor* of n, and n a *multiple* of m. We also say that n is *divisible* by m. If m does not divide n, we write $m \nmid n$. □

Example A.2. *We have* $3 \mid 6$ *because* $6 = 2 \cdot 3$. *Similarly,* $-5 \mid 30$ *because* $30 = (-6) \cdot (-5)$.

Any integer n divides 0, because $0 = n \cdot 0$. The only integer that is divisible by 0 is 0, because $n = 0 \cdot k$ implies $n = 0$. Furthermore, every integer n is divisible by 1, for $n = n \cdot 1$.

Theorem A.3. *For* $m,n,r,s,t \in \mathbb{Z}$ *we have the following:*
(i) If $m \mid n$ *and* $n \mid r$ *then* $m \mid r$.
(ii) If $m \mid n$, *then* $mr \mid nr$ *for all* r.
(iii) If $r \mid m$ *and* $r \mid n$, *then* $r \mid (sm+tn)$ *for all* s,t.
(iv) If $m \mid n$ *and* $n \neq 0$, *then* $|m| \leqq |n|$.
(v) If $m \mid n$ *and* $n \mid m$, *then* $|m| = |n|$.

Proof. [4, p. 3]
The following result is very important. It shows that division with remainder of integers is possible.

Theorem A.4. *If* m *and* n *are integers,* $n > 0$, *then there are uniquely determined integers* q *and* r *such that*

$$m = qn + r \qquad \text{and} \qquad 0 \leqq r < n, \tag{A.5}$$

namely

$$q = \left\lfloor \frac{m}{n} \right\rfloor \qquad \text{and} \qquad r = m - qn. \tag{A.6}$$

The theorem in fact consists of *two* assertions: (i) "There exist integers q and r such that ...", and (ii) "q and r are unique." So we will divide its proof into two parts, proof of *existence* and proof of *uniqueness*.

Proof of Theorem A.4. (i) *Existence.* The numbers m and n are given. Hence we can construct $q = \lfloor m/n \rfloor$, and thus also $r = m - qn$. These are the two equations of (A.6). The last equation is equivalent to $m = qn - r$, which is the first equation of (A.5). The property of the floor bracket implies

$$\begin{aligned} & m/n - 1 < q \leqq m/n && \mid \cdot(-n) \\ \overset{(n>0)}{\Longrightarrow} \quad & n - m > -qn \geqq -m && \mid +m \\ \Longleftrightarrow \quad & n > m - qn \geqq 0. \end{aligned}$$

This implies that $r = m - qn$ satisfies the inequalities $0 \leqq r < n$.

Uniqueness. We now show that if two integers q and r obey (A.5), they also obey (A.6). Let be $m = qn + r$ and $0 \leqq r < n$. Then $0 \leqq r/n = m/n - q < 1$. This implies

$$\begin{aligned} & 0 \leqq \frac{m}{n} - q < 1 && \mid -\frac{m}{n} \\ \Longrightarrow \quad & -\frac{m}{n} \leqq -q < -\frac{m}{n} + 1 && \mid \cdot(-1) \\ \Longrightarrow \quad & \frac{m}{n} \geqq q > \frac{m}{n} - 1 \\ \Longrightarrow \quad & q = \left\lfloor \frac{m}{n} \right\rfloor. \end{aligned}$$

□

To summarize, the proof is divided in two parts: The first one proves that there *exists* two numbers q and r that satisfy (A.5). The second one shows that *if* two numbers q and r satisfying (A.5) also satisfy (A.6). Thus q and r as found in the first part of the proof are unique.

A.3 Searching in unsorted data structures

In this section it is proved that searching one of m entries in an unsorted data structure requires $\Theta(N)$ queries on average, where N denotes the size of the data structure. Such an unsorted data structure may be represented by a database, or a data collection such as an array.

Let U be a given enumerable set, the "universe". Usually $U \subseteq \Sigma^*$ is a subset of all words, or strings, being constructable from a given alphabet Σ. Suppose we wish to search through a finite set $X \subseteq U$ of N elements, the *search space* or *database*. We call X *unsorted* or *unstructured* if there is not imposed any further conditions on X or any order of its elements, i.e., they are considered to be distributed completely by random. Let moreover $S \subset U$ be finite, the *solution set*. Then an *oracle* is denoted as the characteristic function $f : X \to \{0,1\}$ of the solution set S, i.e.,

$$f(x) = \begin{cases} 1 & \text{if } x \in S, \\ 0 & \text{otherwise.} \end{cases} \tag{A.7}$$

In turn, the solution set $S \cap X$ is uniquely determined by the oracle ω, viz., $S \cap X = \{x \in X : \omega(x) = 1\}$. Calling f an oracle we mean that we may have neither access to its internal working, nor immediate access to all argument-value pairs $(x, f(x))$. We only can query it as many times as we like, but with each query comes a computational cost. An oracle does not necessarily *know* the solutions, but it can *recognize* them.[1] Then the SEARCH problem is defined as the problem to find one of the $m = |S|$ items in a database X, given an oracle f which is supposed to be computable in time complexity $T_f(n) = O(n^k)$ for some $k \in \mathbb{N}$ with respect to the maximum length n of a string coding an element in X. Typically, $n = \lceil \log_c N \rceil$ with $N = |X|$ and $c = |\Sigma|$, where Σ is the underlying alphabet.

Theorem A.5. *Let $Q_{N,m}$ denote the number of queries on an unsorted database X with $N = |X|$ entries to find one of m searched elements, but where m is not known to the searcher. Then the expected value $\mathbb{E}[Q_{N,m}]$ of queries is given by*

$$\mathbb{E}[Q_{N,m}] = \begin{cases} N & \text{if } m = 0, \\ \dfrac{N+1}{m+1} & \text{if } m > 0. \end{cases} \tag{A.8}$$

Proof. The case $N > 0$, $m = 0$ is clear, the case $N > 0$, $1 \leqq m \leqq N$ can be proved by induction over N. First, $N = 1$ and $m = 1$ is trivial. For $N > 1$, we notice with Figure A.1

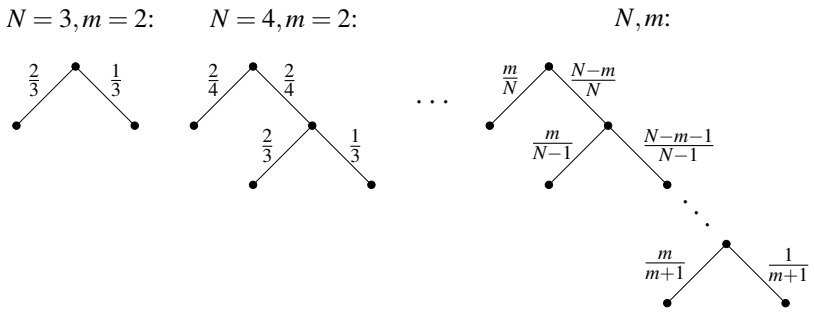


Figure A.1: Probability tree diagrams for each search strategy on an unsorted database with N entries, $m \geqq 0$ of which are marked. Each left branch represents an event of finding a marked item, the last right branch leads to the sure finding in the next step if $m > 0$.

that the first query yields a positive answer with probability $\frac{m}{N}$, and with probability $\frac{N-m}{N}$

[1] It is common in the quantum algorithm literature to implicitly assume the oracle to work efficiently [32]; in complexity theory, however, an oracle (or more precisely, an "oracle Turing machine" M^A for the oracle A) may be a much more general algorithm "transcending worlds" [35, §14.3]. In this sense, an oracle rather plays the role of a "proof checker" or a "verification algorithm" [5, §34.2] in the terminology of complexity theory.

there remain $N-1$ items to be checked. But with formula (A.8) for $N-1$, we then obtain $\mathbb{E}[Q_{N,m}] = \frac{m}{N} + \frac{N-m}{N}\mathbb{E}[1+Q_{N-1,m}] = 1 + \frac{N-m}{m+1} = \frac{N+1}{m+1}$ for $0 < m \leqq N-1$. Thus the case $m = N$ remains to be determined: but it is simply given by $\mathbb{E}[Q_{N,N}] = 1 = \frac{N+1}{N+1}$, i.e., (A.8) holds for all $0 \leqq m \leqq N$. Q.E.D.

Remark A.6. Varying the above problem, we want to search for the position of one of m marked items in an unsorted database with $N = |X| \geqq 2$ entries where we *know* the number m satisfying $0 < m < N$. In other words, we are *guaranteed* a previously known number of marked items. Then we find the position of one of the searched items in

$$\begin{aligned}\mathbb{E}[Q^{\mathrm{pos}}_{N,m}] &= \frac{(N-m)(N-m)!}{(N)_{N-m}} + m\sum_{k=1}^{N-m}\frac{k\,(N-m)_{k-1}}{(N)_k} \\ &= \frac{(N-m)m!}{(N)_m} + \frac{m}{(N)_m}\sum_{k=1}^{N-m} k\,(N-k)_{m-1} \end{aligned} \tag{A.9}$$

queries on average, where $(n)_k := \frac{n!}{(n-k)!}$ for $n, k \in \mathbb{N}$, especially $(n)_0 = 1$, $(n)_n = (n)_{n-1} = n!$. Eq. (A.9) follows directly from Figure A.1. Thus we have $\mathbb{E}[Q^{\mathrm{pos}}_{N,m}] = \Theta(N)$. Some special cases are the following: For $m = 1$, we obtain

$$\mathbb{E}[Q^{\mathrm{pos}}_{N,1}] = \frac{N-1}{N} + \frac{1}{N}\sum_{k=1}^{N-1} k = \frac{N^2+N-2}{2N}. \tag{A.10}$$

For $m = 2$, by $\sum_1^{N-2} k^2 = \frac{(2N-3)(N-2)(N-1)}{6}$, i.e., $\frac{2}{(N)_2}\sum_{k=1}^{N-2} k\,(N-k) = \frac{2}{(N)_2}\left(N\sum_{k=1}^{N-2} k - \sum_{k=1}^{N-2} k^2\right) = \frac{(N-2)(N+3)}{3N}$, we obtain

$$\begin{aligned}\mathbb{E}[Q^{\mathrm{pos}}_{N,2}] &= \frac{2\,(N-2)}{N(N-1)} + \frac{2}{N(N-1)}\sum_{k=1}^{N-2} k\,(N-k) \\ &= \frac{(N-2)(N^2+2N+3)}{3N(N-1)}. \end{aligned} \tag{A.11}$$

For $m = 3$, remembering $\sum_{k=1}^{N-3} k^3 = \binom{N-2}{2}^2$, we have

$$\begin{aligned}\mathbb{E}[Q^{\mathrm{pos}}_{N,3}] &= \frac{6\,(N-3)}{(N)_3} + \frac{3}{(N)_3}\sum_{k=1}^{N-3} k\,(N-k)_2 \\ &= \frac{(N-3)\,[(N-2)(N^2+3N+8)+24]}{4(N)_3} \\ &= \frac{(N+2)\,(N-3)\,(N^2-N+4)}{4(N)_3}. \end{aligned} \tag{A.12}$$

The direct evaluation for higher m is not obvious. Especially, $m = N-1$ yields $\mathbb{E}[Q^{\mathrm{pos}}_{N,N-1}] = \frac{1}{N} + \frac{N-1}{N} = 1$. □

You may ask whether Theorem A.5 is important in computer science. Eventually, all important databases *are sorted*, so the result is irrelevant for usual data applications. But far from it! In fact, *any* database containing datasets with more than one data field is unsorted with respect to at least one field. Take a phone book, containing mainly the name and the corresponding phone number as data fields: any phone book is unsorted with respect to the phone numbers.

Example A.7. *(Searching number in a phone book)* Let $U = \{0, 1, \ldots, 10^{10}-1\}$ be the set of all 10-digit decimal numbers. Consider a phone book $X \subset U$ containing N numbers, and let $S = \{1234567890\}$. Then SEARCH is the decision problem to determine whether the phone number $x_0 = 123$–456–7890 is contained in the phone book. The oracle then is given as

$$f(x) = \begin{cases} 1 & \text{if } x = x_0, \\ 0 & \text{otherwise.} \end{cases}$$

Since a 10-digit number needs $n = \lceil \log_2 10^{10} \rceil = 34$ bits, any number x in X or S, as subsets of Σ^* with $\Sigma = \{0,1\}$, has a length satisfying $|x| \leqq n$. Thus the oracle can work efficiently with at most $n = 34$ steps, comparing successively each possible binary digit. Classically, one needs $\frac{N+1}{2}$ queries on average by Eq. (A.8), whereas Grover's quantum search algorithm requires only $\sqrt{N}$ queries on average. □

Example A.8. *(Known-plaintext attack on a cryptosystem by brute force)* Assume that you have received a plaintext/ciphertext pair of a given cryptosystem and you want to find the secret key. The cryptosystem might be a symmetric cipher, like AES, or a public key cipher, as RSA [6]. They all have in common that their strongness relies on the difficulty to find the key. A brute force attack tries to break the cryptosystem by searching the secret key querying the encryption function successively with all possible keys K until

$$E_K(M) = C,$$

where M is the plaintext and C is the ciphertext. To formulate a brute force attack as a search problem, let $X = U = \{0,1\}^n$ denote the set of all keys of length n bits, and $S = \{K \in U : E_K(M) = C\}$. Then for each $K \in X$, the oracle $f : X \to \{0,1\}$ is given by

$$f(K) = \begin{cases} 1 & \text{if } E_K(M) = C, \\ 0 & \text{otherwise.} \end{cases}$$

By construction of the encryption function, the oracle is polynomial-time with respect to $n = \max\{|K|\colon K \in X\}$. For instance, AES uses keys of length up to $n = 256$ bits, i.e., the search space X contains $N = 2^{256}$ elements. A classical brute force attack thus takes on average about $N/2 = 2^{255}$ steps. Grover's quantum algorithm [17] requires only about $\sqrt{N} = 2^{128}$ steps [7, §6.2.1]. Moreover, as a decision problem the known-plaintext attack is always true in practice, since the ciphertext C has been computed from a given plaintext M, more interesting, of course, is the position of M in the search space X. □

A very important example of a search problem is SAT.

Example A.9. *(SAT)* [35, §4.2] The "satisfiability problem for propositional logic", denoted SAT, asks whether a given Boolean expression $f : \{0,1\}^n \to \{0,1\}$ in conjunctive normal form is satisfiable, i.e., whether there exists an assignment $x = (x_1, \ldots, x_n)$ such that $f(x) = 1$, where 0 denotes `false` and 1 denotes `true`. Here a Boolean expression is a combination of the "literals" x_j and the symbols $\neg$, $\wedge$, $\vee$, (, and). is in conjunctive normal form (CNF) if $f(x) = \bigwedge_{i=1}^m c_i$ where each "clause" c_i is a disjunction of one or more literals x_j or $\neg x_j$ [35, §4.1]. For instance, $f(x_1,x_2) = (x_1 \vee \neg x_2) \wedge \neg x_1$ is satisfiable since $f(0,0) = 1$, whereas

$$f(x_1,x_2,x_3) = x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge (\neg x_1 \vee x_2 \vee x_3)$$

is not satisfiable because $f(x) = 0$ $\forall x \in \{0,1\}^3$. Denote $X = U = \{0,1\}^n$ the space of the 2^n possible assignments to the Boolean formula f, and let $S \subset X$ be the set of all satisfying assignments of f. If f is not satisfiable, S is empty, i.e., $m = |S| = 0$. Then a simple algorithm

to solve this problem is to perform a "brute force" search through the space X and to query f as the oracle function. Since there are $N = 2^n$ possible assignments, SAT may be considered as a special N item search problem with the oracle f working efficiently with time complexity $O(\log^k N)$. Classically, it requires $O(2^n)$ oracle queries on average, whereas Grover's quantum algorithm needs $O(2^{\frac{n}{2}})$ oracle queries on average. □

Example A.10. *(Hamilton cycle problem)* [32, §6.4] A *Hamilton cycle* is a cycle in which each vertex of an undirected graph is visited exactly once. The *Hamilton cycle problem* (HC) is to determine whether a given graph contains a Hamilton cycle or not. Let $X = U$ be the set of all possible cycles beginning in vertex 1, i.e., $x = (x_0, x_1, \ldots, x_{n-1}, x_n)$ where $x_0 = x_n = 1$ and where $(x_1, \ldots, x_{n-1})$ is a permutation of the $(n-1)$ vertices $x_j \neq 1$. In other words, X contains all possible Hamilton cycles which *could* be formed with the n vertices of the graph. Then a simple algorithm to solve the problem is to perform a "brute force" search through the space X and to query the oracle function

$$f(x) = \begin{cases} 1 & \text{if } x \in S, \\ 0 & \text{otherwise,} \end{cases} \tag{A.13}$$

where S is the solution set of all cycles of the graph,

$$S = \{x \in U : (i_{j-1}, i_j) \in E \ \forall j = 1, \ldots, n\}. \tag{A.14}$$

If the graph does not contain a Hamilton cycle, then S is empty and $m = |S| = 0$. The oracle only has to check whether each pair (x_{j-1}, x_j) of a specific possible Hamilton cycle is an edge of the graph, which requires time complexity $O(n^2)$ since $|E| \leqq n^2$; because there are n pairs to be checked in this way, the oracle works with total time complexity $O(n^3)$ per query. (Its space complexity is $O(\log_2 n)$ bits, because it uses E and x as input and thus needs to store temporarily only the two vertices of the considered edge, requiring $O(\log_2 n)$.)

Since there are at most $N = (n-1)! = O(n^n) = O(2^{n \log_2 n})$ possible orderings, the Hamilton cycle problem is a special N item search problem. Classically, it requires $O(2^{n \log_2 n})$ oracle queries on average, whereas Grover's quantum algorithm needs $O(2^{\frac{n}{2} \log_2 n})$ oracle queries on average [7, §6.2.1].

According to Dirac's Theorem, any graph in which each vertex has at least $n/2$ incident edges has a Hamilton cycle. This and some more such *sufficient* criterions are listed in [9, §8.1]. □

A problem being apparently similar to the Hamilton cycle problem is the Euler cycle problem. Its historical origin is the problem of the "Seven Bridges of Königsberg", solved by Leonhard Euler in 1736.

Example A.11. *(Euler cycle problem)* [32, §3.2.2] Let $\Gamma = (V, E)$ be an undirected graph consisting of n numbered vertices $V = \{1, \ldots, n\}$ and the edges $E \subseteq V^2$ such that $(x,x) \notin E$ and $(x,y) \in E$ implies $(y,x) \in E$ for all $x, y \in V$. An *Euler cycle* is a closed-up sequence of edges, in which each edge of the graph is visited exactly once, If we shortly denote $(x_0, x_1, \ldots, x_m)$ with $x_0 = x_m = 1$ for a cycle, then a necessary condiiton to be Eulerian is that $m = |E|$. The *Euler cycle problem* (EC) then is to determine whether a given graph contains an Euler cycle or not. By Euler's theorem [9, §0.8], a connected graph contains an Euler cycle if and only if every vertex has an even number of edges incident upon it. Thus EC is decidable in $O(n^3)$ computational steps, counting for each of the n vertices x_j in how many of the at most $\binom{n}{2}$ edges (x_j, y) or $(y, x_j) \in E$ it is contained. As a search problem, the search space X consists only of the n vertices of the considered graph, and the answer is known after at most n countings of the edges incident on each vertex. □

For each of these search problems, an oracle function is known which is polynomially computable with respect to n, i.e., which has time complexity $O(\log^k N)$ and is thus efficient.

Appendix B

Dictionary for mathematics and computer science

A

array Array, Feld *(wörtl.: Schlachtreihe)*
artificial neural network künstliches neuronales Netz
assertion Behauptung
assume annehmen
assumption Annahme

B

bound Schranke, Grenze
business information systems Wirtschaftsinformatik

C

calculus Differential- und Integralrechnung
circuit board Platine, Leiterplatte
complete vollständig
computer science Informatik
column Spalte *(auch einer Matrix)*

D

deduce herleiten
denominator Nenner
digets Auszug, Abriss
die, *pl.* **dice** Würfel
disjoint disjunkt
divisor Teiler, Divisor

E

economic order quantity Losgröße
edge Kante
equation Gleichung
equilateral triangle gleichseitiges Dreieck
even number gerade Zahl
evenly divisible ohne Rest teilbar
extract the (*n*-th) root die (*n*-te) Wurzel ziehen

F

feedback Rückkopplung
finite endlich
fraction Bruch

G

gcd ggT (größter gemeinsamer Teiler)
greatest common divisor größter gemeinsamer Teiler

H

hash feinhacken; vermasseln, verhunzen
heap Heap (*wörtl.* Haufen), Halde
hence deshalb, also
[the equation] holds [die Gleichung] gilt

IJK

intersection Schnittmenge
induction assumption Induktionsannahme
induction start Induktionsanfang
induction step Induktionsschritt
infinite unendlich
inflection point Wendepunkt *(math. Kurvendiskussion)*
initial value Anfangswert
insert einsetzen
integer ganze Zahl
invertible umkehrbar
isoscele triangle gleichschenkliges Dreieck

L

lattice *math* Gitter
lcm kgV (kleinstes gemeinsames Vielfache)
least common multiple das kleinste gemeinsame Vielfache
let be ... sei ...
(straight) line Gerade
linked list verkettete Liste
load factor Füllfaktor, Füllgrad *(e-r Hashtabelle)*
lot size Losgröße
lower bound = lower limit untere Schranke, untere Grenze
lowercase letter Kleinbuchstabe

M

mapping Abbildung
merge verschmelzen, zusammenführen
motherboard Hauptplatine
multicriterion optimization Mehrkriterienoptimierung

N

neural network neuronales Netz
node Knoten
numerator Zähler

O

objective function Zielfunktion
obtain erhalten
obvious offensichtlich, klar
odd number ungerade Zahl

P

perpendicular senkrecht
plane *math* Ebene
pointer Pointer, Zeiger
polygon Polygon, Vieleck
polyhedron Polyeder, Vielflächner
polynomial Polynom; polynomial
potential set Potenzmenge
preimage Urbild *(e-r Abbildung)*
prime number Primzahl
proof Beweis
proposition *log* Aussage; *math* Satz, Lehrsatz
prove beweisen

Q

queue (Warte-)Schlange

R

reciprocal value Kehrwert
record Record, Datensatz
remainder Rest
rational number rationale Zahl
real number reelle Zahl
(*n*-th) root (*n*-te) Wurzel (*to extract* - ziehen)
row (Matrix-) Zeile

S

sales figures Verkaufszahlen
satisfy the equation die Gleichung erfüllen, der Gleichung genügen
scalene triangle ungleichseitiges Dreieck
scatterplot Punktwolke, Streudiagramm
self-loop Schlinge *(math)*
in the sequel im folgenden
sequence Folge *(math)*
series Reihe *(math)*
set Menge
slack variable Schlupfvariable (beim Simplexalgorithmus)
spot Ort; Fleck; (Spiel-, Würfel)Auge
stack Stack (*wörtl.* Stapel)
suffice genügen
sufficient condition hinreichende Bedingung
suppose annehmen
subscripted letters indizierte Buchstaben
subset Teilmenge
subtree Teilbaum

T

tetrahedron Tetraeder
therefore daher
thread einfädeln, aufreihen; Faden, *comp* Thread
thus so, also, deshalb
time series Zeitreihe
toss (hoch)werfen; Wurf
total of the digits of Quersumme von
triangle Dreieck

UVW

up to a constant bis auf eine Konstante
upper bound = upper limit obere Schranke, obere Grenze

uppercase letter Großbuchstabe
vertex Ecke, Eckpunkt; Knoten (eines Graphen)

XYZ

yield ergeben

Appendix C

Arithmetical operations

Fundamental operations of arithmetic – Grundrechenarten

Operation	English	German
$x = y$	x equals y	x ist gleich y
$x \approx y$	x is approximately equal to y	x ist ungefähr gleich y
$x \leqq y$	x is less/smaller than or equals y	x ist kleiner gleich y
$x \geqq y$	x is greater than or equals y	x ist größer gleich y
$x + y$	x plus y	x plus y
$x - y$	x minus y	x minus y
$x \cdot y$	x times y	x mal y
$2 \cdot 3 = 6$	two threes are six	2 mal 3 ist (gleich) 6
x/y	x divided by y, x over y	x (geteilt) durch y
x^y	x to the (power of) y	x hoch y
x^{-y}	x to the minus y	x hoch minus y
x^2	x squared	x zum Quadrat
x^3	x cubed	x hoch 3
x^4	x to the 4th (power)	x hoch 4
x^5	x to the 5th (power)	x hoch 5
$\vdots$	$\vdots$	$\vdots$
$\sqrt{x}$	square root of x	Wurzel (aus) x
$\sqrt[3]{x}$	cube root of x	dritte Wurzel aus x
$\sqrt[n]{x}$	nth root of x	n-te Wurzel aus x
$n!$	n factorial	n Fakultät
$\binom{n}{m}$	choose m out of n	n über m, m aus n
$\lfloor x \rfloor$	floor of x	untere Gaußsche Klammer von x
$\lceil x \rceil$	ceiling of x	obere Gaußsche Klammer von x
$\forall$	for all	für alle
$\exists$	there exists	es existiert

Bibliography

[1] A. P. Barth. *Algorithmik für Einsteiger*. Vieweg, Braunschweig Wiesbaden, 2003.

[2] J. Bather. *Decision Theory. An Introduction to Dynamic Programming and Sequential Decisions*. John Wiley & Sons, Chichester, 2000.

[3] C. H. Bennett. 'Logical Depth and Physical Complexity'. In R. Herken, editor, *The Universal Turing Machine. A Half-Century Survey*, pages 207–235. Springer-Verlag, Wien, 1994.

[4] J. A. Buchmann. *Introduction to Cryptography*. Springer-Verlag, New York, 2001.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, New York, 2nd edition, 2001.

[6] A. de Vries. 'The ray attack on RSA cryptosystems'. In R. W. Muno, editor, *Jahresschrift der Bochumer Interdisziplinären Gesellschaft eV 2002*, pages 11–38, Stuttgart, 2003. ibidem-Verlag. `http://arxiv.org/abs/cs/0307029`.

[7] A. de Vries. *Quantum Computation. An Introduction for Engineers and Computer Scientists*. Books On Demand, Norderstedt, 2012.

[8] A. de Vries and V. Weiß. 'Grundlagen der Programmierung'. Vorlesungsskript, Hagen, 2014. `http://www4.fh-swf.de/media/java.pdf`.

[9] R. Diestel. *Graphentheorie*. Springer-Verlag, Berlin Heidelberg, 2. edition, 2000.

[10] W. Domschke and A. Drexl. *Einführung in Operations Research*. Springer-Verlag, Berlin Heidelberg, 5th edition, 2002.

[11] T. Ellinger, G. Beuermann, and R. Leisten. *Operations Research. Eine Einführung*. Springer-Verlag, Berlin Heidelberg, 4th edition, 1998.

[12] M. Falk, R. Becker, and F. Marohn. *Angewandte Statistik mit SAS. Eine Einführung*. Springer-Verlag, Berlin Heidelberg, 2nd edition, 1995.

[13] O. Forster. *Analysis 1*. Vieweg, Wiesbaden, 9. edition, 2008.

[14] D. Fudenberg and J. Tirole. *Game Theory*. MIT Press, Cambridge, 1991.

[15] I. Gerdes, F. Klawonn, and R. Kruse. *Evolutionäre Algorithmen. Genetische Algorithmen – Strategien und Optimierungsverfahren – Beispielanwendungen*. Vieweg, Wiesbaden, 2004.

[16] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, 1994.

[17] L. K. Grover. 'Quantum mechanics helps in searching for a needle in a haystack'. *Phys. Rev. Lett.*, 79(2):325, 1997.

[18] H. P. Gumm and M. Sommer. *Einführung in die Informatik*. Oldenbourg Verlag, München, 2008.

[19] R. H. Güting. *Datenstrukturen und Algorithmen*. B.G. Teubner, Stuttgart, 1997.

[20] S. Hackel, T. Schäfer, and W. Zimmer. *'Praktische Sicherheitskonzepte'*. Verlag Werner Hülsbusch, Boizenburg, 2.3 edition, 2010. urn:nbn:de:0008-20100305103.

[21] D. Harel and Y. Feldman. *Algorithmik. Die Kunst des Rechnens*. Springer-Verlag, Berlin Heidelberg, 2006.

[22] J. Havil. *Verblüfft?!* Springer-Verlag, Berlin Heidelberg, 2009.

[23] V. Heun. *Grundlegende Algorithmen*. Vieweg, Braunschweig Wiesbaden, 2000.

[24] D. W. Hoffmann. *Theoretische Informatik*. Carl Hanser Verlag, München, 2009.

[25] M. J. Holler and G. Illing. *Einführung in die Spieltheorie*. Springer-Verlag, Berlin Heidelberg New York, 3. edition, 1996.

[26] F. Kaderali and W. Poguntke. *Graphen, Algorithmen, Netze. Grundlagen und Anwendungen in der Nachrichtentechnik*. Vieweg, Braunschweig Wiesbaden, 1995.

[27] S. C. Kleene. 'Turing's Analysis of Computability, and Major Applications of It'. In R. Herken, editor, *The Universal Turing Machine. A Half-Century Survey*, Wien, 1994. Springer-Verlag.

[28] D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, 3rd edition, 1997.

[29] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, Reading, 3rd edition, 1998.

[30] S. O. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. Teubner, Wiesbaden, 2005.

[31] Z. Michalewicz. *Genetic Algorithms + Data Structure = Evolution Programs*. Springer Verlag, Berlin Heidelberg, 3rd edition, 1996.

[32] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2000.

[33] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, Heidelberg Berlin, 4. edition, 2002.

[34] F. Padberg. *Elementare Zahlentheorie*. Spektrum Akademischer Verlag, Heidelberg Berlin, 2. edition, 1996.

[35] C. M. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[36] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C++. The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2nd edition, 2002.

[37] B. Schneier. *Angewandte Kryptographie*. Addison Wesley, Bonn, 1996.

[38] R. Sedgewick. *Algorithmen in C*. Addison-Wesley, Bonn, 1992.

[39] K. Sydsæter and P. Hammond. *Mathematik für Wirtschaftswissenschaftler*. Pearson Studium, München, 2004.

[40] H. Tempelmeier. *Material-Logistik. Modelle und Algorithmen für die Produktionsplanung und -steuerung in Advanced-Planning-Systemen*. Springer-Verlag, Berlin Heidelberg New York, 2006.

[41] A. Törn and A. Zilinskas. Global Optimization. *Lecture Notes in Computer Science*, 350, 1989.

[42] L. von Auer. *Ökonometrie. Eine Einführung*. Springer-Verlag, Berlin Heidelberg, 3. edition, 2005.

[43] K. Weicker. *Evolutionäre Algorithmen*. B. G. Teubner, Stuttgart Leipzig Wiesbaden, 2002.

[44] N. Wirth. *Algorithmen und Datenstrukturen*. B.G. Teubner, Stuttgart Leipzig, 1999.

Links

1. *http://www.nist.gov/dads/* – NIST Dictionary of Algorithms and Data Structures
2. *http://riot.ieor.berkeley.edu/* – RIOT Optimization Testbed,

Index

$\lll$, 47

$\mathbb{R}^+$, 18
gcd, 17

action, 77
acyclic, 67
adjacency list, 65
adjacency matrix, 64
adjacent, 64
Akra-Bazzi theorem, 30
algebra, 5
algorithm, 15
    Dijkstra's -, 61, 73
    Euclid's -, 10, 28
    evolutionary -, 96
    Floyd-Warshall -, 61, 71
    genetic -, 61, 97
    greedy -, 61
    recursive -, 25
    simplex -, 61, 88
    Wagner-Whitin -, 61, 82
allele, 97
alphabet, 44
ant colony optimization, 62
artificial intelligence, 97
assignment, 12
asymptotically tightly bounded, 20
average-case analysis, 21
Axelrod's genetic algorithm, 106

backwards induction, 79
balanced divide and conquer algorithm, 36
Bellman functional equation, 79
Bellman's Optimality Principle, 72, 77
BFS algorithm, 65
bit rotation, 47
Boolean expression, 13
brute force, 97
bucket, 46
bucket sort, 36

candidate solution, 56
capacity, 46
chromosome, 97
circular left-shifting, 47
class diagram, 66
collision, 46, 49
combinatorial optimization, 57
common divisor, 16
comparison sort algorithm, 35
complexity, 17
complexity class, 18
condition, 13
conjugate, 93
constraint, 57
constraints
    primary, 87
control structure, 12
correctness, 16
cost function, 77
crossover, 98, 105
cryptography, 112
cu, 80
currency unit, 80
cycle, 67

decision, 72, 76, 77
decision problem, 68, 79
decision space, 79
decision vector, 77
depth-first search, 67
digit, 19
Dijkstra algorithm, 73
Dijkstra's algorithm, 61
discrete optimization, 57
distance, 70
divide and conquer, 36
divides, 108
divisor, 108
dominates, 18
duality, 92
dynamic programming, 72, 75

edge, 63
efficient, 21
elite principle, 99
empty word, 45
Euclidean algorithm, 10, 28
Euclidean space, 57
Euler cycle, 68, 113
evolution, 97
evolution strategy, 61, 96
evolutionary
    algorithm, 61, 96
    programming, 61, 96
exception, 14
exchange, 12
exhaustion, 44, 61, 75
exhaustive, 97
expansion
    b-adic -, 19
exponential time complexity, 21
extended Euclidean algorithm, 28

factorial, 25
Fibonacci heap, 74
fitness function, 98
floor-brackets, 9
Floyd-Warshall algorithm, 61, 71

game theory, 105
Gauß-brackets, 9
gene, 97
generation, 99
genetic algorithm, 61, 97
genetic operator, 98
genotype, 97
golden ratio, 15
graph, 63
    weighted -, 69
Gray code, 102
greatest common divisor, 10, 17
greedy algorithm, 61

Hamilton cycle, 113
Hamiltonian cycle, 67, 68
Hamiltonian cycle problem, 68
Hamming distance, 101
hash table, 46
hash value, 46
hash-function, 46
hashing, 48
HashMap, 44
HashSet, 44
HC, 68
heuristic, 106
Huffman code, 61
hyperspace, 57

independent, 53
individual, 97
input, 15
insertion sort, 34
instruction, 12
integers, 108
ISBN, 46

key comparison sort algorithm, 35
knowledge, 97, 103

Landau symbol, 18
lattice, 57
law of motion, 77
learn, 106
length, 64, 70
letter, 44
linear optimization problem, 88
linear probing, 53
linear programming, 88
load factor, 51
loci, 97
logarithmic time complexity, 21
loop, 13

Master theorem, 30
maximum problem, 56
MD5, 48
minimum problem, 56, 92
modulo, 9
Monte Carlo technique, 96
multi-criterion optimization, 60
multiple, 108
multistage decision problem, 79
mutation, 98

Nash equilibrium, 104
natural numbers, 108
negative cycle, 70
neighbours, 64
NP-complete, 85

objective function, 87
operation, 11
optimization, 56
optimization problem, 68
optimization problems, 75
optimum path, 75
oracle, 110
oracle function, 68
output, 15
OX operator, 103

Pareto optimization, 60
particle swarm optimization, 62
path, 64
    shortest -, 70
permutation, 85, 103
pigeonhole sort, 36
pivot element, 38, 89
polynomial, 20
polynomial time complexity, 21
population, 97
premature convergence, 101
primary constraints, 87
priority queue, 73
programming
    dynamic, 75
    linear, 88
pseudocode, 10

qu, 79
quantity unit, 79

Rastrigin function, 59
recombination, 98
rectangle rule, 89
recurrence equation, 29, 30
recursion, 25
recursion equation, 30
recursion step, 26
recursion tree, 28
recursive call, 26
regression, 55
relation, 63
relaxation, 71
repetition, 13
return, 12
Rivest, Ron, 47

roulette-wheel selection, 99
running time, 20, 22

SAT, 106, 112
search space, 56
selection, 13, 98
selectionSort, 34
self-loop, 64
sequence, 13
sequential decision problem, 79
SHA, 47
shortest path, 70
simple path, 64
simplex, 92
simplex algorithm, 61, 88
simplex tableau, 88
simulated annealing, 96
single-source shortest path, 70
slack variable, 88
solution space, 96
stage, 75
state, 75
storage balance equation, 81
strategy, 105
string, 44
subroutine, 14
swarm intelligence, 61

tableau, 88
time, 10
time series, 55
tournament selection, 99
transition function, 77
transition law, 77
transpose, 88, 93
traveling salesman problem, 55, 103
travelling salesman problem, 84
triangle inequality, 85
truncation selection, 99
TSP, 55, 84, 103
Turing machine, 15

Unicode, 45
uniform, 53
unit
    currency, 80
    quantity, 79
universe, 45

variable
    slack, 88
vertex, 63

Wagner-Whitin algorithm, 61, 82
walk, 64
weighted graph, 69
word, 44
worst-case analysis, 21