Grundlagen der Informatik

Dieter Pawelczak

Start in die Technische Informatik

Leseprobe des im Shaker-Verlag erschienenen Buchs: ISBN 978-3-8322-7540-2

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek vezeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Prof. Dr.-Ing. D. Pawelczak
Start in die Technische Informatik

Lehrbuch begleitend zur Vorlesung *Grundlagen der Informatik*, 4 TWS, 3 ECTS
Fakultät für Elektrotechnik und Technische Informatik
Universität der Bundeswehr München, 85577 Neubiberg
Zi. 5104/Geb. 41

Printed in Germany.

ISBN 978-3-8322-7540-2
ISSN 0945-0807

Shaker Verlag GmbH • Postfach 101818 • 52018 Aachen
Telefon: 02407 / 95 96 - 0 • Telefax: 02407 / 95 96-9
Internet: www.shaker.de • E-Mail: info@shaker.de

Inhaltsverzeichnis

1 Einführung

1.1 Umfeld und Struktur

Das vorliegende Buch *Start in die Technische Informatik* dient als Begleitskript für die Vorlesung *Grundlagen der Informatik* im Bachelor Studiengang *Technische Informatik und Kommunikationstechnik*. Das Lernziel ist, ein Verständnis für die grundlegenden, meist englischsprachigen Begriffe der Informatik zu schaffen und die Zusammenhänge und Abläufe der Rechnertechnik begreifbar zu machen.

Die folgenden Kapitel geben eine ausführliche Einführung in das breite Themengebiet der Informatik:

- *Kapitel 2: Theoretische Grundlagen*: Algorithmen, Darstellung und Kodierung von Daten im Rechner, Zustandsdiagramme, Ablaufsteuerung, Programmiersprachen und Softwareentwicklung.
- *Kapitel 3: Aufbau und Funktionsweise von Rechnern*: Digitale Logik- und Speicherbausteine, Rechnerarchitekturen, Zentraleinheit, Optimierung von Mikroprozessoren, Bussysteme, Peripherie und Betriebssysteme.
- *Kapitel 4: Kommunikation und Rechnernetze*: Topologien und Funktionsweise von Rechnernetzen, TCP/IP-Protokoll, Netzwerkdienste und IT-Sicherheit.
- *Kapitel 5: Anwendungen*: Datenbanken, Office-Anwendungen.

Es werden die grundlegenden Konzepte gezeigt, welche zum Entwickeln von Algorithmen erforderlich sind und wie sich diese auf einen Rechner umsetzen lassen. Auf das konkrete Programmieren in einer Programmiersprache wird hierbei jedoch verzichtet. Vielmehr werden Algorithmen in einer graphischen Darstellung und einer leicht verständlichen deutschen Beschreibung dargestellt, um auch Studierenden ohne Vorkenntnisse der Informatik die Einführung leicht zu gestalten.

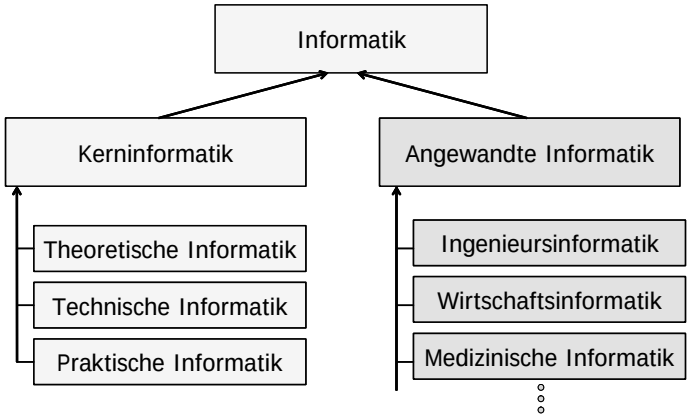
Zahlen werden in der im Bereich der Computertechnik üblichen Notation dargestellt: Nachkommastellen werden durch einen Dezimalpunkt getrennt. Das Komma kommt dagegen bei Aufzählungen zum Einsatz. Algorithmen oder Programmtexte werden in `Courier-Schrift` dargestellt. Diagramme erfolgen – sofern nicht anders bezeichnet – in der UML (engl. *unified modeling language*). Kapitel 6 bietet eine Aufgabensammlung, dessen ausführliche Lösungen auf der Webseite der zugehörigen Lehrveranstaltung zu finden sind[1]. Literaturangaben sind am Ende der Abschnitte zu finden. Hinweise auf Webseiten sind als Fußnoten aufgeführt.

Die verwendeten Software- und Hardwarebezeichnungen sind (wenn auch nicht immer ausdrücklich gekennzeichnet) in der Regel eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

1. http://www.unibw.de/etti6/pawelczak/lehre/GDI/

1.2 Einführung in die Informatik

Die Informatik ist die Wissenschaft von der systematischen und automatisierten Verarbeitung von Informationen. Der Begriff Informatik ist ein Kunstwort aus den Begriffen Information und Automation. Die Informatik hat sich Anfang der 70er Jahre von einem Teilgebiet der Mathematik zu einer eigenständigen Wissenschaft etabliert. Hierfür waren insbesondere Arbeiten auf dem Gebiet der theoretischen Informatik ausschlaggebend, welche die Entwicklung von Programmiersprachen, Compilern, Betriebssystemen in dieser Zeit enorm vorangetrieben haben. Eine etwas breitere Definition der Informatik kommt von der Gesellschaft für Informatik[2]: Informatik ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Information. Diese schließt die unter Kap. 1.1 genannten Themen, insbesondere die der Rechnerarchitekturen und Rechnernetze, mit ein. Grundsätzlich ist jedoch die Informatik auch eine Wissenschaft der Abstraktion, da es gilt, das richtige (optimale) Modell für ein Problem zu finden und davon eine angemessene automatisierbare Technik abzuleiten.



Kerninformatik

Die Informatik wird in die *Kerninformatik* und *Angewandte Informatik* gegliedert. Die Kerninformatik lässt sich dabei weiter unterteilen in:

Theoretische Informatik

◊ *Theoretische Informatik*: Sie behandelt die theoretischen Grundlagen, wie Algorithmen, Modellierung, Kodierung, Datenstrukturen, formale Sprachen, Automatentheorie und mathematische Grundlagen.

Technische Informatik

◊ *Technische Informatik*: Sie beschäftigt sich mit Hardware: Rechnerarchitekturen, Speicherbausteinen, Prozessoren, Bussystemen, Peripheriegeräten etc.

Praktische Informatik

◊ *Praktische Informatik*: Hier steht die Software-Entwicklung im Vordergrund. Das Tätigkeitsfeld umfasst insbesondere die Entwicklung von Compilern, Betriebssystemen, graphischen Benutzeroberflächen und Anwendungsprogrammen.

Angewandte Informatik

Die Angewandte Informatik wendet die Methoden der Kerninformatik zur Lösung spezifischer Nutzerprobleme an. Sie hat praktisch in allen Bereichen unseres modernen Lebens Einzug gehalten: Wir nutzen sie beim Telefonieren (mobil, digital und analog), Autofahren, Geschirr- oder Wäschewaschen, Musikhören, Segeln, bei allen bargeldlosen Transaktionen, in jedem Büro, in jeder Arztpraxis, in jedem Supermarkt, sogar in unserem elektronischen Reisepass usw. Neben diesen, uns zum Teil sichtbaren oder verborgenen Entwicklungen der Informatik haben sich interdisziplinäre Forschungsbe-

2. http://www.gi-ev.de/

reiche der Angewandten Informatik entwickelt, so z. B. die Wirtschaftsinformatik, die Medizinische Informatik, die Bioinformatik und die Ingenieurinformatik.

Die Informatik ist aus den Ingenieurwissenschaften nicht mehr wegzudenken und stellt neben der Mathematik und der Physik eine der tragenden Säulen der Ingenieurwissenschaften dar. Für die Elektrotechnik stellt die Informatik nicht nur die notwendigen Werkzeuge für den Schaltungsentwurf, die Schaltungssimulation sowie für die gesamte Automatisierung des Fertigungsprozesses elektronischer Systeme bereit, sondern realisiert mit der Digitaltechnik eine neue Klasse elektronischer Schaltungen, welche weitestgehend die traditionelle analoge Schaltungstechnik verdrängt hat. Als Beispiel sei das *Software Defined Radio* (SDR) genannt: Anstelle eines traditionellen analogen Radio-Empfängers wird das Funksignal der Antenne (ggf. mit einer vorgeschalteten Mischerstufe, um in einen niedrigeren Frequenzbereich zu gelangen) einem schnellen Analog-Digital-Wandler zugeführt. Die gesamte Empfangsstufe wird dann digital ausgeführt. Ein SDR kann damit theoretisch jedes beliebige komplizierte Funksignal, wie z. B. ein OFDM-Signal (engl. *orthogonal frequency division multiplex*) genauso wie ein klassisches FM-Signal (engl. *frequency modulation*) dekodieren. Nach der digitalen Dekodierung kann der Signalinhalt, wie z. B. ein Sprachsignal, mit Hilfe eines Digital-Analog-Wandlers an einen Ausgangsverstärker ausgegeben oder digital über Bluetooth an ein Headset übertragen werden.

Geschichte und Bedeutung

Die Geschichte der Informatik kann Jahrtausende umfassen, wenn man das Algorithmisieren, d.h., die schrittweise Lösung von Problemen als Startpunkt der Informatik auffasst: In diesem Fall ist der Ursprung in der Mathematik zu finden. Bereits *Euklid* beschreibt in seinem Werk *Die Elemente* um ca. 325 v. Chr. den Euklidischen Algorithmus zum Finden des größten gemeinsamen Teilers zweier Zahlen[3]. Auch wenn in der Antike einfache Rechenapparaturen bekannt waren, wie. z. B. die Multiplikation mit Hilfe von Zahnrädern, so fehlt dieser „Informatik der Antike“ der Bezug zur Automatisierung im Sinne unseres Informatikbegriffs. Diese wird in Form von mechanischen Rechenmaschinen intensiv ab dem 17. Jahrhundert untersucht (*Blaise Pascal, Gottfried Wilhelm Leibniz*), wobei Leibniz bereits die Nutzung des dualen Zahlensystems untersucht hat.

Als Meilenstein und damit vielleicht als Geburtsstunde der Informatik ist die Erfindung der Lochkarte zu nennen. Diese wurde 1728 von dem französischen Mechaniker *Jean Baptiste Falcon* und später von *Joseph-Marie Jacquard* für die Erzeugung von individuellen Webmustern bei mechanischen Webstühlen eingesetzt und bis in die 60er Jahre des letzten Jahrhunderts als Speichermedium für Programme und Daten genutzt. Ein weiterer Vorreiter der Informatik ist *Charles Babbage*, welcher um 1837 ein erstes modernes Rechnerkonzept mit Rechenwerk, Zahlenspeicher sowie Ein- und Ausgabeeinheit vorstellte. Obwohl seine *Analytical Engine* nie gebaut wurde, befasste sich *Ada Lovelace* intensiv mit der Maschine und beschrieb, wie sie zu programmieren sei, um Bernoulli-Zahlen berechnen zu können. Sie gilt damit als erste Informatikerin der Geschichte.

Konrad Zuse griff in den 30er Jahren das Konzept von Babbage auf. Der Durchbruch gelang ihm 1942 mit seinem Relaisrechner Z3, welcher als der erste funktionstüchtige Computer der Welt gilt. Ebenso von Babbage inspiriert, baute *Howard Aiken* an der Harvard University in den USA die elektromechanische Rechenmaschine Mark I. Die strikte Trennung von Daten- und Programmspeicher wird noch heute als Harvard-Architektur bezeichnet.

3. Der Begriff Algorithmus wurde jedoch erst um 825 n. Chr. geprägt.

Zu dieser Zeit behauptete der IBM-Chef Thomas J. Watson Sr. angeblich: "Ich glaube, es gibt einen weltweiten Bedarf an vielleicht fünf Computern". Eine solche Aussage ist angesichts des enormen technischen und personellen Aufwands für Installation, Bedienung und Wartung der Rechenmaschinen zu dieser Zeit realistisch. So hat sich beispielsweise aus dieser Zeit der Begriff *Bug* (engl. für Wanze) für einen Fehler in einem Computerprogramm etabliert: Ein verirrtes Insekt genügte, die mechanischen Relaiskontakte zu stören und damit den korrekten Betrieb des Rechners zu unterbrechen.

Erst der Einzug der Transistortechnik und die Integration von Transistorschaltungen auf Siliziumwafer haben entscheidend dazu beigetragen, dass Größe und Preis moderner Rechner einem Massenmarkt gerecht wurden. Nicht zu vernachlässigen ist die enge Verzahnung der Elektrotechnik mit der Informatik: Wurden erste Mikroprozessoren noch „von Hand“ entworfen (z. B. Intel 4004, 1971, ca. 2300 Transistoren), so ist bei der heutigen Integrationsdichte (z. B. AMD Opteron K10, 2007, ca. 463 Mio. Transistoren) ein Computer-gestützter Entwurf nicht mehr wegzudenken.

Die Informatik hat somit selbst zu der raschen Entwicklung moderner Rechner beigetragen: Die Hardware-Beschreibung komplexer Mikroprozessoren kann heute für die Entwicklung sog. SOCs (engl. *system on chip*) als Software-Bibliothek aus dem Internet heruntergeladen werden[4]. Wir beschreiben nicht mehr das Layout (Schaltplan) von integrierten Schaltungen, sondern verwenden eine Beschreibungssprache, um die Funktionalität der Hardware zu definieren. Ausgehend von dieser Beschreibung lässt sich die eigentliche Transistorschaltung und das Layout automatisch generieren. Gleichzeitig können wir entsprechende Testvorschriften einführen, um den Entwurf automatisch zu testen.

Entgegen der prognostizierten fünf Computer weltweit verfügen heute etwa 70 % der deutschen Privathaushalte (entspricht etwa 25 Mio. Haushalte) über einen oder mehrere Computer[5].

Dabei ist diese Zahl noch klein verglichen mit der Vielzahl an eingebetteten Systemen (engl. *embedded systems*), welche durch eine geeignete Programmierung

- ◊ die Funktionalität in alltägliche Geräte wie Handys, Uhren, Kameras, MP3-Player, Kühlschränke, Waschmaschinen etc. bringen,
- ◊ die Motorsteuerung in Kraftfahrzeugen übernehmen,
- ◊ für Komfort und Sicherheit in Fahrzeugen, Schiffen und Flugzeugen sorgen usw.

Zudem greift jeder Bürger bewusst oder unbewusst auf die Rechenleistung anderer Organisationen zu: Beispielsweise bei der Nutzung digitaler Dienste wie Internet oder Mobilfunk, beim Einkaufen im Laden oder über das Internet, am Geldautomaten oder sei es bei den Grundbedürfnissen, wie der Versorgung mit Wasser, Gas und Strom.

4. siehe z. B. http://www.opencores.org/
5. http://www.heise.de/newsticker/Fast-70-Prozent-der-Haushalte-haben-einen-Computer--/meldung/78835 vom 29.09.2006

1.3 Literatur

[Bau04] Friedrich L. Bauer: *Informatik – Führer durch die Ausstellung*. Deutsches Museum München, Neuauflage 2004

[Gall07] Jens Gallenbacher: *Abenteuer Informatik: IT zum Anfassen; von Routenplaner bis Online-Banking*. München: Elsevier / Spektrum Akad. Verl., 2007

[Reg08] Gerard O'Regan: *A brief history of computing*. London: Springer, 2008

Weiterführende Lehrbücher für die Vertiefung des Stoffs Grundlagen der Informatik:

[Gum04] H. P. Gumm, M. Sommer: *Einführung in die Informatik*. München [u.a.]: Oldenbourg 2006 (7., vollst. überarb. Aufl.)

[Hero07] Helmut Herold, Bruno Lurz, Jürgen Wohlrab: *Grundlagen der Informatik: praktisch - technisch - theoretisch*. München [u.a.]: Pearson Studium 2007

[Rem03] Ulrich Rembold, Paul Levi: *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. München [u.a.]: Hanser 2003 (4. Aufl.)

Interessante Links:

[heise] C. Heise, A. Heise, C. Persson (Hrsg.): *Heise Online: news Meldungen des Tages*. Online: http://www.heise.de

[leo] *Englisch-Deutsches Online-Wörterbuch.* Online: http://dict.leo.org

[wikip] Wikimedia Foundation Inc.: *Wikipedia – Die freie Enzyklopädie*. Online: http://de.wikipedia.org

1.4 Danksagung

Meiner Frau Alexandra und meinen beiden Kindern Michael und Stephanie möchte ich ganz herzlich für ihr großes Verständnis danken, da ich viele Stunden – insbesondere auch an den Wochenenden – nur über diesem Manuskript verbracht habe. Mein Dank gilt den Mitarbeitern der WE6 an der *Fakultät für Elektrotechnik und Technische Informatik* (ETTI) der Universität der Bundeswehr München. Insbesondere danken möchte ich Frau Gisela Müller und Herrn Robert Klopp für das Lektorat sowie meinem Kollegen Friedrich Sösemann für seine vielen Kommentare und Ideen. Zuletzt möchte ich mich bei den Studenten unserer Fakultät ETTI aus den Jahrgängen 2006 und 2007 für die zahlreichen Hinweise bei der Erstellung des Buchs bedanken.

Neubiberg, im August 2008

2 Theoretische Grundlagen

2.1 Information, Daten und Algorithmen

Information

Daten

Die *Information* stellt die zentrale, immaterielle Grundgröße der Informationstechnik dar. Der Begriff *Information* beinhaltet einen Neuigkeitsgehalt und eine Verwendbarkeit und grenzt sich damit von dem Begriff *Daten* ab. Der Informationsgehalt von Daten hängt unmittelbar mit deren Verarbeitbarkeit zusammen. Unter Daten kann man auch die „Digitalisierung der Information“ verstehen.

Beispiel 2.1: *Information und Daten*

Eine MP3-Datei enthält (Audio-) Informationen und unterscheidet sich von anderen digitalen Daten dadurch, dass eine geeignete Anwendung die Information verarbeiten, also wiedergeben kann. Verfügt ein System nicht über eine geeignete Anwendung, dann kann auf die Information nicht zugegriffen werden. Die Daten stehen weiterhin zur Verfügung und können eingesehen werden, z. B. durch das Öffnen der MP3-Datei mit einem Texteditor(!).

Semantik

Die Daten und die zugehörige Bedeutung (*Semantik*) bzw. Interpretation der Daten stellen die Information dar. Unterschieden wird zwischen einfachen Datentypen, wie z. B. Zahlenwerten, Zeichenfolgen und komplexen, zusammengesetzten Datentypen, wie z. B. Personendaten (Name, Geburtsdatum, Personalnummer usw.), vgl. Kap. 2.2.

Der Zusammenhang zwischen Daten, Information und Algorithmen nach [Vogt04] ist in Bild 2.1 dargestellt.

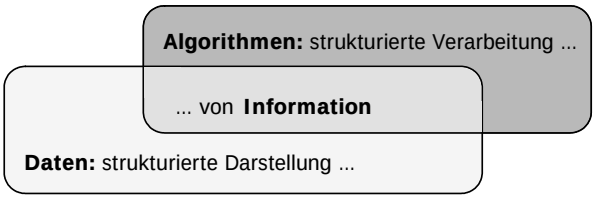
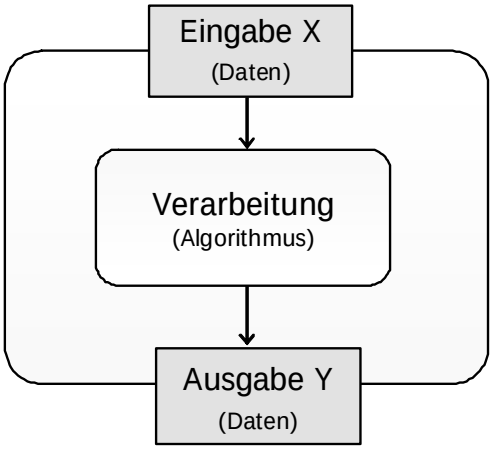


Bild 2.1: *Daten, Information und Algorithmen*

Das klassische Hauptprinzip der Informationsverarbeitung besteht aus den drei Schritten: Eingabe, Verarbeitung und Ausgabe („EVA-Prinzip“). Im oben genannten Beispiel steht für die Eingabe die MP3-Datei, für die Verarbeitung die Umformung der komprimierten Daten in Audio-Informationen und für die Ausgabe die Audioschnittstelle des Rechners. Die Grundlagen für die Verarbeitung von Informationen geben *Algorithmen*.

Bild 2.2: *EVA-Prinzip*



Der Algorithmus zur Wiedergabe der MP3-Datei kennt die Semantik der Daten. Folglich werden Informationen verarbeitet. Man spricht daher von Informationsverarbeitung.

Die Einheit Bit

Die kleinste darstellbare Information benötigt zwei Zustände. Andernfalls wäre der Inhalt konstant und damit vorab bekannt. Es würde folglich der Neuigkeitsgehalt fehlen. Beispiele für zwei mögliche Zustände sind:

Bit

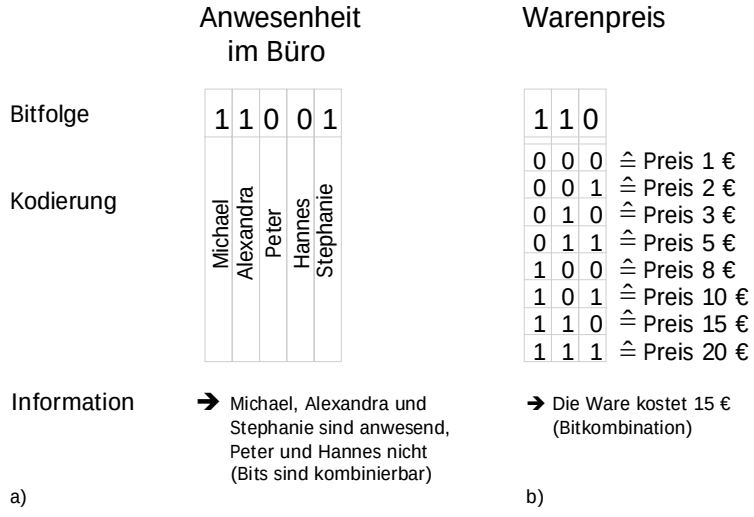
nein - ja	0 V - 5 V	0 - 1
falsch - wahr	0 V - 1,8 V	false - true
weiß - schwarz	0 mA - 20 mA	kurz - lang

Dualsystem

Kodierung, Code

Diese kleinstmögliche Einheit der Information wird *Bit* (engl. *bi*nary digi*t*, für Binärziffer) bezeichnet. Die Kombination mehrerer Bits zu einer Bitfolge erlaubt die Speicherung beliebiger Daten, sofern die Bedeutung der Bits festgelegt wird. Diese Festlegung wird Kodierung oder Code genannt. Bild 2.3 zeigt zwei Beispiele für Bitfolgen und eine mögliche Kodierung.

Bild 2.3:
Beispiele für die Kodierung von Bitfolgen:
a) Anwesenheit
b) Preisinformation



Bei der Kodierung können den einzelnen Bits Bedeutungen zugewiesen werden (Beispiel für Anwesenheit) oder einer gesamten Bitfolge (Beispiel für Preise). Viele Kodie-

rungen kombinieren diese beiden Aspekte, wie z. B. der ASCII-Code (engl. *american standard code for information interchange*)[1]. Diese weitverbreitete Kodierung definiert, wie Ziffern und Buchstaben auf einem Rechner gespeichert werden. Ein sieben Bit langer Binärcode beschreibt die Semantik der einzelnen Zeichen: Großbuchstaben, Kleinbuchstaben, Satzzeichen, Sonderzeichen, Zahl usw. Erst mit Hilfe einer solchen Kodierung ist der Datenaustausch zwischen mehreren Rechnern möglich: Sofern zwei Rechner den ASCII-Code kennen, lässt sich, z. B. beim Versenden einer E-Mail, eine Textnachricht auf einem Rechner erzeugen und auf dem anderen Rechner korrekt darstellen.

In diesem Zusammenhang sei auf das von *Samuel Morse* entwickelte Morse-Alphabet hingewiesen. Es stellt auch eine Kodierung von Zeichenfolgen dar, welche nur korrekt gesendet und empfangen werden kann, wenn der Sender und Empfänger über ausreichende Kenntnisse des Morse-Alphabets verfügen.

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 ––··	0 –––––
1 ·––––	2 ··–––	3 ···––	4 ····–	5 ·····	6 –····	7 ––···	8 –––··	9 ––––·

Bild 2.4: *Morse-Alphabet*

Oktett, Byte

Eine Bitfolge (oder der Zusammenschluss) von 8 Bits wird Oktett bzw. *Byte* (engl. *binary term*[2], für Binärausdruck) bezeichnet. Eindeutige Begriffe für größere Bitfolgen existieren nicht[3], so dass üblicherweise die Bezeichnungen 16-Bit, 32-Bit, 64-Bit etc. Anwendung finden. Die Größeneinheiten im binären System beziehen sich auf die Basis zwei. Ein KByte entspricht daher nicht etwa 1000 Bytes, sondern $2^{10} = 1024$ Bytes[4], vgl. Tabelle 2.1.

Tabelle 2.1: *Binäre Größeneinheiten KByte, MByte, GByte, TByte*

Einheit	Umrechnung	Größe in Bytes
1 KByte	2^{10} = 1024 Byte	1024 $\approx 10^3$
1 MByte	2^{20} = 1024 KByte	1048576 $\approx 10^6$
1 GByte	2^{30} = 1024 MByte	1073741824 $\approx 10^9$
1 TByte	2^{40} = 1024 GByte	1099511627776 $\approx 10^{12}$

1. vgl. Bild 2.22
2. ursprünglich von engl. *bite* – dt. Happen/Bissen, also das, was ein Rechner auf einmal „schlucken“ kann.
3. Oft findet man die Bezeichnung *Word*, welche sich auf die Wortbreite des Zentralprozessors bezieht. Ein Word auf einem 16-Bit-Prozessor entspricht folglich 16 Bit, auf einem 32-Bit-Prozessor 32 Bit. Allerdings ist es gerade im PC-Bereich üblich, mit einem *Word* 16 Bit zu bezeichnen, da der ursprüngliche 8086-Prozessor eine Wortbreite von 16 Bit aufwies. *DWord* (*double word*) entspricht dann auf einem PC 32 Bit und *QWord* (*quad word*) 64 Bit.
4. Es gibt viel Streit um die Bezeichnungen Kilobyte, Megabyte, Gigabyte und Terabyte, da 1 k (Kilo) bei den SI-Einheiten dem Faktor 1000 (1 Mega = 10^6, 1 Giga = 10^9, 1 Tera = 10^{12}) entspricht. Häufig verwendet man daher anstelle der Einheitenvorsätze „Kilo“, „Mega“, „Giga“, „Tera” die Bezeichnungen „K“, „M“, „G“, „T“, um auf die Basis 2 aufmerksam zu machen. Es ist also nicht verwunderlich, dass man Festplatten mit 200 Gigabyte kauft, aber das Betriebssystem nur 186,2 GByte meldet, oder dass die Kapazität eines 4,7-GB-DVD-Rohlings tatsächlich nur 4,38 GByte beträgt. Im Rahmen dieses Skripts beziehen sich KByte und Kilobyte immer auf die Basis zwei. Vorschläge des IEC (IEC 60027-2: 2005) für die Bezeichnungen Kibibyte, Mebibyte, Gibibyte, Tebibyte haben sich in der Praxis bisher nicht durchgesetzt (das *bi* steht für die Basis 2).

MSB, LSB

Für jegliche Art der Kodierung ist es erforderlich, eine feste Anordnung (Bitwertigkeit) bei Bitfolgen festzulegen. Dafür werden das höchstwertigste Bit MSB (engl. *most significant bit*) und das niederwertigste Bit LSB (engl. *least significant bit*) definiert. Bei der üblichen Schreibweise werden die höherwertigen Bits (in Analogie zu den Dezimalstellen in der mathematischen Notation) links gesetzt:

Beispiel 2.2: *MSB und LSB eines Oktetts*

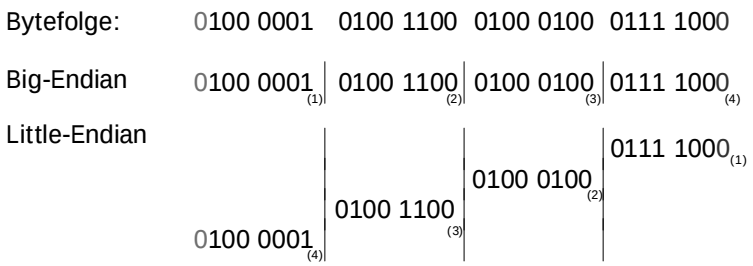
```
  0 1 0 0 1 1 0 0
MSB             LSB
```

Byte-Reihenfolge
Big-Endian

Little-Endian

Werden längere Bitfolgen zu Bytefolgen zusammengefasst, dann gibt es zwei unterschiedliche Möglichkeiten der Anordnung: *Big-Endian* und *Little-Endian*. Bei Big-Endian ist die Reihenfolge von den höherwertigen („Big“) Bytes zu den niederwertigen. Big-Endian entspricht also unserer Schreibweise von links nach rechts. Bei Mikroprozessoren kommt häufig die Little-Endian-Anordnung (vom niedrigsten Byte zum höchstwertigsten Byte) zur Anwendung. Dabei liegen das LSB auf der niedrigsten und das MSB auf der höchsten Byteposition.

Bild 2.5: *Big- und Little-Endian: Die Nummerierung in Klammern weist auf die Reihenfolge der Bytes hin.*



Little-Endian wird von der Intel-x86-Architektur, den Alpha-Prozessoren und einigen Mikrocontrollern, wie z. B. die Siemens-C166-Familie, Siemens-Tricore, verwendet. Das Big-Endian-Format wird von den Motorola-68k-, Coldfire- und PowerPC-Prozessoren unterstützt sowie für die Speicherung von Daten in Java und bei der Übertragung von Daten in Rechnernetzen (Internet) eingesetzt. Einige Prozessoren erlauben eine softwaremäßige Konfiguration und unterstützen beide Formate, wie z. B. ARM- und PowerPC-Prozessoren.

Die Big- und Little-Endian-Darstellung führt insbesondere bei der Portierung von Software sowie bei der Kommunikation zwischen unterschiedlichen Rechnerplattformen zu Problemen.

Übliche, im Bereich der Rechnertechnik verwendete Kodierungen werden in Kap. 2.2 ausführlich behandelt.

Der Informationsgehalt

Jedes Bit erlaubt die Kodierung von zwei Zuständen. Mit zwei Bits lassen sich folglich 2·2 = 4, mit 3 Bits 2·2·2 = 8 und mit 4 Bits 16 Zustände kodieren. Wir erkennen die Rechengrundlage:

$$n = 2^k,$$

wobei eine Anzahl von *k* Bits *n* mögliche Zustände (Kombinationen) ergibt.

Wenn wir wissen wollen, wie viele Bits wir für eine gegebene Menge von *n* Zuständen benötigen, müssen wir die Formel umkehren. Die Umkehrung einer Potenz ergibt sich aus dem Logarithmus zur entsprechenden Basis. Aufgrund der zwei elementaren Zustände eines Bits müssen wir als Basis die Zahl Zwei wählen. Der Logarithmus zur Basis zwei wird *Logarithmus Dualis* (*ld*) genannt. Er erlaubt die Berechnung der notwendigen Bits *k* für eine vorgegebene Menge *n*:

Logarithmus Dualis

$$ld(n) = \frac{\log n}{\log 2} = k$$

Natürlich machen Nachkommastellen bei der Einheit Bit keinen Sinn, wenn Datenleitungen für die Informationsübertragung oder Zellen für die Informationsspeicherung verwendet werden müssen. Das Ergebnis muss folglich aufgerundet werden. Die folgenden Aussagen ergeben sich beispielhaft für *k*:

Tabelle 2.2: *Notwendige Anzahl von Bits zur Speicherung unterschiedlicher Mengen*

Menge *n*	***k* = ld(*n*)**	**Notwendige Anzahl an Bits**
Eine Dezimalziffer ($n = 10$)	$k = 3.32$	4 Bit
Lateinisches Alphabet ($n = 26$)	$k = 4.70$	5 Bit
3 Dezimalstellen (n=1000)	$k = 9.97$	10 Bit
10 Dezimalstellen ($n = 10^{10}$)	$k = 33.22$	34 Bit
1000 Dezimalstellen ($n = 10^{1000}$)	$k = 3321.93$	3322 Bit

Bei einer gleichverteilten Auftrittswahrscheinlichkeit aller in der Menge enthaltenen *n* Zustände entspricht der Logarithmus Dualis dem *Informationsgehalt* der vorgegebenen Menge. Allerdings ist die Berechnung des Informationsgehalts über den Logarithmus Dualis nur bei Gleichverteilung gültig. Es entscheiden die Auftrittswahrscheinlichkeiten der einzelnen Zustände einer Menge über die erforderliche Bitanzahl, die zur Übertragung erforderlich ist. So hat *Claude Shannon* den Informationsgehalt $I(x)$ eines einzelnen Zeichens *x* über dessen Auftrittswahrscheinlichkeit $p(x)$ definiert:

Informationsgehalt

$$I(x) = ld\frac{1}{p(x)} = -ld(p(x))$$

Für n=10 Zeichen ergibt sich bei einer Gleichverteilung, d.h. einer jeweiligen Auftrittswahrscheinlichkeit von 10% in die Formel eingesetzt: $I(x) = 3.322$. Tritt eines der Zeichen nahezu kontinuierlich, z. B. mit 99 % auf, so wird der Informationsgehalt verschwindend gering: $I(x) = 0.0145$. Dies entspricht der Definition von Information: Würde ein Zeichen konstant auftreten, so geht der Informationsgehalt gegen null, da

der Neuigkeitsgehalt fehlen würde. Ein Zeichen, das nur sehr selten auftritt, weist dagegen einen sehr hohen Informationsgehalt auf. Dies ist in Bild 2.6 dargestellt: Ein weißes Blatt Papier, welches als Informationsspeicher dienen soll, enthält keine Information. Ein einzelner schwarzer Punkt weist dagegen einen hohen Informationsgehalt auf, wie z. B. dessen Größe oder Position.

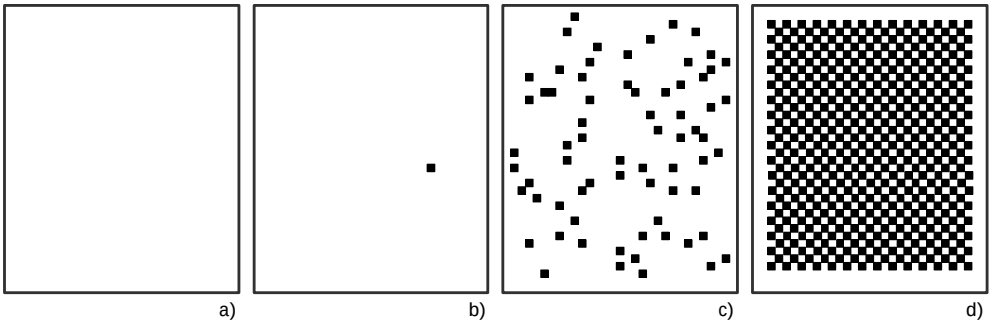


Bild 2.6:
Informationsgehalt
a) leeres weißes Blatt,
b) mit schwarzem Punkt,
c) mit vielen schwarzen Punkten
d) mit Schachbrettmuster

Die effiziente Speicherung der Information setzt eine Kenntnis der Auftrittswahrscheinlichkeiten der einzelnen Zeichen voraus. Es wäre sicherlich unsinnig, anstelle des schwarzen Punkts nach Bild 2.6 b), die weißen Stellen zu speichern. Vielmehr würde man nur die Größe und die Position des schwarzen Punkts speichern. Mit steigender Auftrittswahrscheinlichkeit der schwarzen Punkte wird jedoch auch diese Kodierung ineffizient. Sinnvoller ist für Bild 2.6 c) die Speicherung der Abstände zwischen den Punkten, da diese erheblich kleiner sind als die absoluten Koordinaten und sich deshalb mit weniger Bits pro Punkt abspeichern lassen. Man nennt diese Art der Kodierung *Lauflängenkodierung*. Sie wird typischerweise bei der Fax-Übertragung angewandt, da in der Regel ein Brief aus mehr weißen als schwarzen Stellen besteht. Die Lauflängenkodierung ist völlig ungeeignet für Bild 2.6 d). Hier lässt sich die Information höchst effizient durch die Beschreibung des Musters, z. B. dem Schachbrettmuster mit 27 Feldern pro Zeile und 33 Zeilen speichern. Eine Beschreibung der einzelnen 891 Punkte ist dabei nicht erforderlich. Vielmehr wird sogar deutlich, dass jetzt ein einzelner Punkt erheblich weniger Information trägt als der einzelne Punkt in Bild 2.6 b).

Lauflängenkodierung

Datenkompression

Sofern eine Kodierung Auftrittswahrscheinlichkeiten oder Muster berücksichtigt, kann eine Datenkompression erfolgen: Zur Speicherung werden weniger Bits, als für die Datendarstellung erforderlich sind, benötigt. Eine typische Anwendung ist die Kompression von Bildern. Bild 2.7 ist mit 256 Graustufen (acht Bits pro Pixel) gespeichert. Bei einer Auflösung von 400 dpi (engl. *dots per inch*, siehe S. 157) und einer Größe von ca. 9,5 x 6,35 cm werden insgesamt 1516 · 1000 = 1516000 Bytes (ca. 1,44 MBytes) benötigt. Verschiedene Kompressionsverfahren reduzieren die Größe der Bilddatei jedoch erheblich. Eine Lauflängenkodierung bewirkt bereits eine Einsparung von etwa 10 %. Allerdings weist die Aquarellzeichnung auch nur wenige im Farbton konstante Flächen auf. Das im Internet weit verbreitete PNG-Format (engl. *portable network graphics*)[5] ermöglicht es, das Bild mit 953 KBytes zu speichern; es benötigt nur ca. 66 % Speicher verglichen mit dem Rohbild. Das komprimierte Bild speichert anstelle der einzelnen Grauwerte sich wiederholende Zeichenfolgen. Zusätzlich wird die Häufigkeit der Zeichenfolgen untersucht: Häufige Zeichenfolgen werden mit weniger Bits kodiert als seltene Zeichenfolgen. Außerdem wird eine typische Eigenschaft für Bilddaten genutzt, dass benachbarte Punkte sich in der Regel nur geringfügig unterscheiden.

5. siehe http://tools.ietf.org/html/rfc2083

Bild 2.7: *Graustufenbild (Stuttgart Staatstheater, Aquarell, A. Kraner 1993)*

Die tatsächliche Speichermenge, die für die abgedruckte Darstellung von Bild 2.7 erforderlich ist, beträgt 258 KByte. Dies entspricht 18 % der Pixeldaten. Für diese hohe Kompressionsrate genügt eine reine Kodierung nicht. Zusätzlich erfolgt eine Datenreduktion. Man spricht von einer verlustbehafteten Datenkompression, da die ursprünglichen Daten im Hinblick auf eine höhere Kompression verändert worden sind. Das bekannteste Verfahren bei der Bilddatenreduktion ist das JPEG-Verfahren[6] (engl. *joint photographic experts group*). Das JPEG-Verfahren verzichtet auf die exakte Reproduktion von Details. Der Gesamteindruck des Bildes bleibt aber bei großen Kompressionsraten erhalten.

verlustbehaftete Datenkompression

Die zur Übertragung des Bildes erforderliche Bitmenge kann aber noch erheblich weiter reduziert werden: Ist dem Empfänger das Bild bekannt, genügt beispielsweise der Verweis auf eine Katalognummer oder den Bildtitel.

Die Eigenschaften von Algorithmen

Algorithmen beschreiben in detaillierter und eindeutiger Vorschrift die schrittweise Lösung eines Problems. Ein typisches Beispiel für einen alltagsüblichen Algorithmus ist das Rezept zum Backen eines Kuchens: *250 g Butter, 4 Eier nach und nach dazugeben und schaumig schlagen, 500 g Mehl unterheben usw.* Ein weiteres einfaches Beispiel für einen Algorithmus ist die Addition zweier größerer Zahlen im Kopf: *Addiere die einzelnen Stellen von rechts und addiere ggf. einen Übertrag von der vorausgehenden Operation*.

Die Beschreibung von Algorithmen kann formal durch eine Beschreibungssprache, Pseudo-Code, eine Programmiersprache oder graphisch erfolgen. Wir verwenden zur Veranschaulichung eine Textbeschreibung sowie die graphische Darstellung mit Hilfe

Darstellung von Algorithmen

6. siehe http://www.jpeg.org/ – der Name JPEG weist auf die gemeinsame Standardisierung durch ISO und ITU-T hin. Das Verfahren wird von den beiden Standards ISO/IEC IS 10918-1 und ITU-T Recommendation T.81 beschrieben.

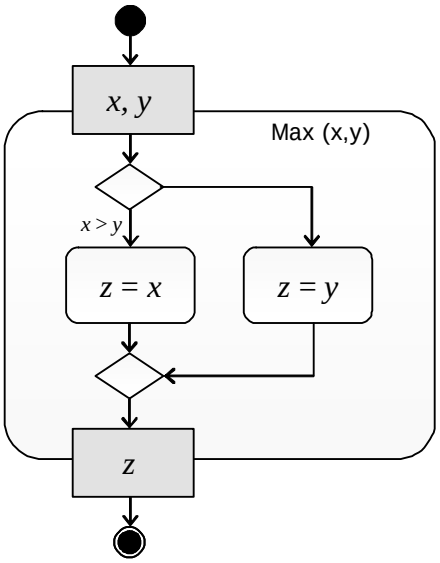
der UML (engl. *unified modeling language*)[7].In dieser Darstellungsform kommen drei Kontrollstrukturen zum Einsatz, vgl. Bild 2.8:

1. *Verarbeitungsschritt*: Der einzelne Verarbeitungsschritt als mathematischer Ausdruck, Zuweisung oder Anwendung eines weiteren Algorithmus wird als *abgerundetes Rechteck* dargestellt.
2. *Verzweigung*: Die Verzweigung aufgrund einer vorgegebenen Bedingung wird durch eine *Raute* repräsentiert.
3. *Zusammenführung*: Die Zusammenführung zweier Pfade aufgrund einer vorangegangenen Verzweigung erfolgt in der Darstellung ebenfalls als *Raute*.

Weiterhin wird der gesamte Algorithmus in einem abgerundeten Rechteck dargestellt. Zusätzlich können Ein- und Ausgabeparameter im Randbereich als Rechtecke hinzugefügt werden, wie beim EVA-Prinzip in Bild 2.2 gezeigt ist. Der Start wird mit einem schwarzen Punkt, das Ende mit einem umrahmten Punkt und der Ablauf mit Pfeilen gekennzeichnet. Bei einer Zuweisung entspricht das Gleichheitszeichen nicht einer Gleichsetzung im mathematischen Sinne: Vielmehr wird der Wert einer Berechnung in einer Variablen gespeichert. Die Zuweisung $x = x+1$ bewirkt folglich, dass der Wert von x um eins erhöht wird. Variablen repräsentieren den „Speicher“ eines Algorithmus. Sie können während der Verarbeitung unterschiedliche Werte annehmen. An einem einfachen Beispiel – dem Finden des Maximalwertes zweier Zahlen x und y – sei die Textbeschreibung und die graphische Darstellung gezeigt, vgl. Bild 2.8.

Bild 2.8: *Algorithmus zum Finden des Maximums zweier Zahlen X und Y*

```
Maximum von x, y

Falls x > y:
   z = x
sonst:
   z = y

Ausgabe: z
```



Algorithmen sind im Allgemeinen:

Eigenschaften von Algorithmen

- ◊ *diskret*: die Ausführung erfolgt in einzelnen Schritten durch einfache Grundoperationen,
- ◊ *statisch finit*: endlich in der Notation, d.h., die Beschreibung muss endlich sein,
- ◊ *dynamisch finit*: endlich in den Ressourcen (während der Ausführung). Ein Algorithmus darf z. B. nicht unendliche Zahlenfolgen für die Verarbeitung benötigen,
- ◊ *vollständig*: neben der vollständigen Beschreibung des Lösungsweges müssen auch Spezialfälle erfasst werden,

7. Die Darstellung entspricht einer Teilmenge des Aktivitätsdiagramms der UML 2.0, siehe Kap. "Die Grundelemente imperativer Programme und ihre Darstellung" auf S. 57.

◊ *terminiert*: endlich in der Abarbeitung, d.h., die Anzahl der Einzelschritte ist während der Ausführung begrenzt,

◊ *determiniert*: bei gleichen Eingaben erfolgen gleiche Ausgaben (Wiederholbarkeit),

◊ *deterministisch*: zu jedem Zeitpunkt der Ausführung sind die Operationen und Operanden festgelegt und der Folgeschritt (Ablauf) eindeutig vorherbestimmt.

Bild 2.9 stellt einige der Kerneigenschaften graphisch dar.

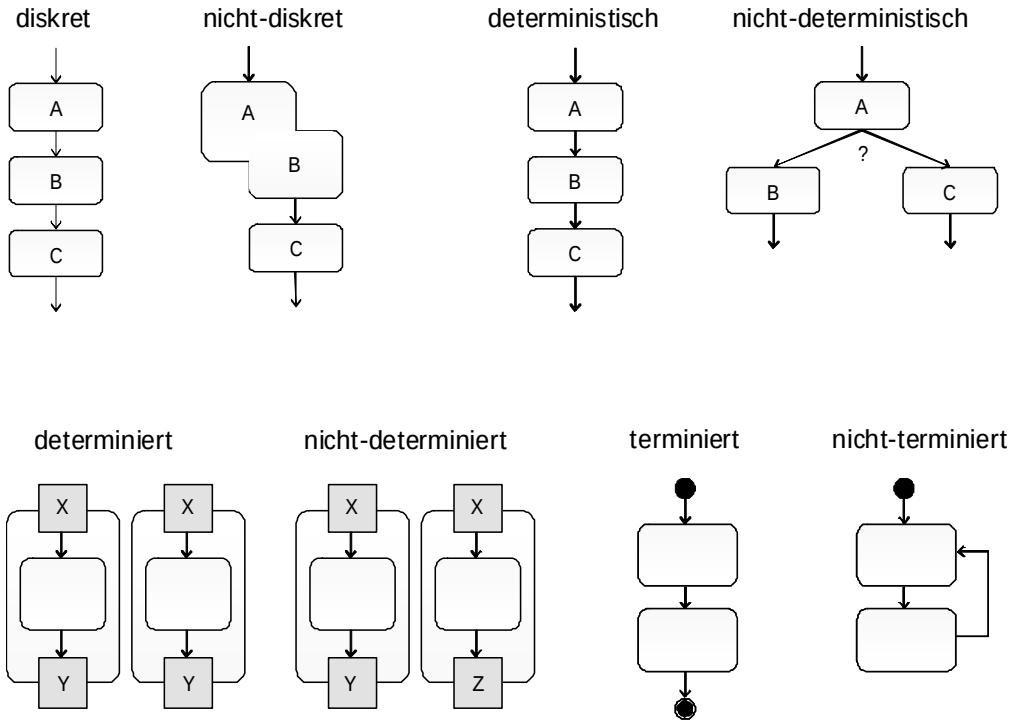


Bild 2.9: *Eigenschaften von Algorithmen*

◊ Ein Algorithmus besteht aus diskreten Schritten. Ein nicht-diskretes System würde dagegen aus fließenden Übergängen zwischen den einzelnen Operationen bestehen.

◊ Bei einem deterministischen System sind die Reihenfolge der Schritte und die Inhalte der Schritte vorgegeben. Ein nicht-deterministischer Algorithmus würde zufällig Folgeschritte wählen oder mit zufälligen Werten arbeiten[8].

◊ In der Regel liefern Algorithmen auf die gleiche Eingabe die gleiche Ausgabe. Es leuchtet ein, dass ein Algorithmus zur Mittelwertbildung von Prüfungsnoten bei gleichen Noten auch immer den gleichen Mittelwert bildet. Sofern unterschiedliche Ausgaben bei den gleichen Eingabedaten erfolgen, handelt es sich um einen nicht-determinierten Algorithmus.

◊ Weiterhin wird von einem Algorithmus erwartet, dass er nach einer endlichen Zahl an Arbeitsschritten ein Ergebnis zu der Eingabe liefert. Allerdings wird auch die endliche Zahl an Schritten einer sinnvollen Bearbeitungszeit gegenübergestellt: Algorithmen, welche eine exponentiell oder gar faktoriell wachsende Bearbeitungszeit gegenüber den Eingabedaten aufweisen, gelten genauso als nicht-terminiert wie Algorithmen, die gewollt oder ungewollt in einer Endlosschleife verweilen.

8. Eine zufällige Reihenfolge in der Verarbeitung kann natürlich auch das Ergebnis von zufälligen Werten sein: Die Entscheidung, ob in Bild 2.9 Schritt B oder Schritt C auf Schritt A folgt, kann z. B. durch ein Bit gefällt werden, was einen zufälligen Wert aufweist.

An einem Beispiel sollen die Eigenschaften von Algorithmen nun untersucht werden. Das zu lösende Problem ist die Berechnung der Quadratwurzel. Zunächst sehen wir uns die mathematische Definition der Quadratwurzel an [Bro91]:

$$\sqrt{x} = y, (x > 0, y^2 = x)$$

deklarativ

Diese mathematische Definition wird *deklarativ* genannt, da sie die Wurzelfunktion zwar eindeutig über das Quadrat definiert, aber keinen Lösungsweg anbietet. Eine direkte Umformung in einen Algorithmus kann deshalb nicht erfolgen. Wir benötigen ein

imperativ

imperatives[9] Verfahren, wie z. B. das Newtonsche Iterationsverfahren:

Beispiel 2.3: *Deterministischer Algorithmus zur Berechnung der Quadratwurzel mit Hilfe des Newtonschen Iterationsverfahrens*

Quadratwurzel von x

Falls $x \leq 0$:
    terminiere

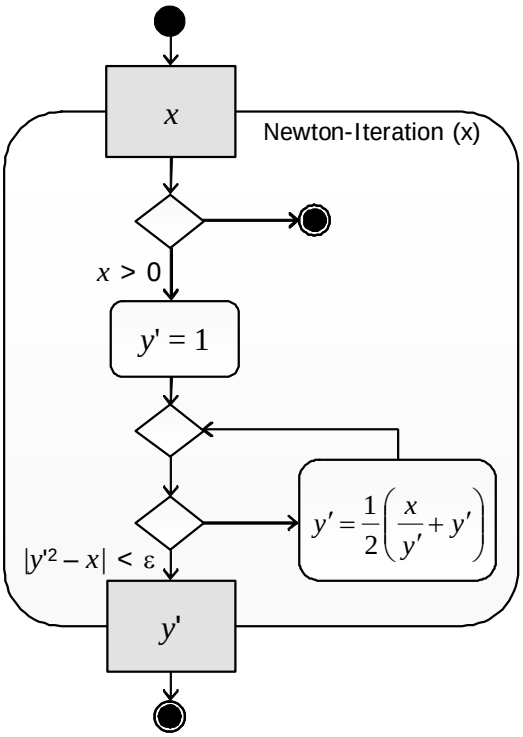
Startwert y'=1

Wiederhole bis $|y'^2 - x| < \varepsilon$ mit einem hinreichend kleinen ε (Restfehler) gelöst wird[10]:

    Berechne neuen Schätzwert y'
    aus dem Mittelwert von: y' und $\frac{x}{y'}$

Ausgabe: y'

Bild 2.10: *Graphische Darstellung des Newtonschen Iterationsverfahrens*



9. In [Abel01] findet man die prägnanten Definitionen: deklarativ: „Was ist“ und imperativ „Wie geht das“.

10. Die Abbruchbedingung enthält die deklarative Definition; sie ist also Teil des Algorithmus.

Eine Analyse des Algorithmus nach den oben genannten Kriterien ergibt:

Beispiel 2.4: *Analyse der Eigenschaften*

Er ist diskret *(besteht nur aus der Quotienten- und Summenbildung, dem Vergleich und den Zuweisungen),* statisch finit *(da Sie sonst nicht über das Lesen des Algorithmus hinausgekommen wären),* dynamisch finit *(es wird nur Speicher für den Schätzwert y' und die Zwischenrechnung benötigt, obwohl der Algorithmus ggf. mehrfach wiederholt wird),* vollständig *(der Wertebereich von x wird auf Spezialfälle überprüft),* terminiert *(es ist eine Abbruchbedingung enthalten[11]),* determiniert *(es erfolgen bei gleicher Eingabe die gleichen Ergebnisse),* deterministisch *(die Reihenfolge ist eindeutig festgelegt, ebenso bleibt die Art der Operationen konstant).*

Zuletzt wollen wir uns die schrittweise Lösung des Problems $y = \sqrt{2}$ ansehen:

Tabelle 2.3: *Newtonsche Iteration zur Berechnung der Quadratwurzel*

Schätzwert	Quotient	Mittelwert = neuer Schätzwert
1	2/1 = 2	Mittelwert (1,2) = 1.5
1.5	2/1.5 = 1.33333	Mittelwert (1.5, 1.3333) =1.41667
1.41667	2/1.41667 = 1.41176	Mittelwert (1.41667, 1.41176) = 1.41421

Das Beispiel zeigt, dass die Quadratwurzel bereits nach drei Iterationen[12] auf fünf Nachkommastellen genau berechnet werden kann.

Rekursion

An dieser Stelle sei eine weitere Klasse der Algorithmen aufgeführt: die *rekursiven* Algorithmen. Die *Rekursion* ist ein mächtiges Werkzeug zur Beschreibung von Algorithmen, sofern sich der Algorithmus als eine Wiederholung seiner selbst darstellen lässt. Wir betrachten eine Lösung zur Berechnung der Fakultät:

Beispiel 2.5: *Rekursiver Algorithmus zur Berechnung der Fakultät*

```
Fakultät von n:

  falls n = 1:
       Ausgabe: 1

  sonst:
       Ausgabe: n * (Fakultät von n - 1)
```

Die Wahl der Rekursion als Berechnungsvorschrift führt zu einer impliziten Zerlegung des Problems in *n*-1 Teilprobleme (= Teilmultiplikationen). Gleichzeitig bleibt der Algorithmus einfach zu lesen, da auf komplizierte Konstrukte wie Wiederholungen bzw. Schleifen verzichtet wird. Ein wichtiger Bestandteil rekursiver Algorithmen ist die Abbruchbedingung, welche eine unendliche erneute Abarbeitung des Algorithmus verhindert und für eine Terminierung sorgt. In Bsp. 2.5 wird die Abbruchbedingung erfüllt, wenn `n = 1` ist. Die Berechnung der Fakultät von 5 führt beispielsweise zu der Folge:

11. Allerdings erfordert die Terminiertheit zwei weitere Voraussetzungen, damit die Abbruchbedingung erfüllt werden kann: Der Algorithmus liefert tatsächlich einen Näherungswert der Quadratwurzel und der Restfehler ist größer als es die Rechengenauigkeit unseres Rechners zulässt.

12. Mit Iterationen bezeichnen wir in der Informatik Wiederholungen/Schleifen.

Beispiel 2.6: *rekursive Verarbeitung des Fakultätsalgorithmus*

Fakultät von 5 =
5 · *Fakultät von* 4 =
5 · 4 · *Fakultät von* 3 =
5 · 4 · 3 · *Fakultät von* 2 =
5 · 4 · 3 · 2 · *Fakultät von* 1 =
5 · 4 · 3 · 2 · 1 =
120

Für die Lösung eines Problems ist häufig ein rekursiver Algorithmus leichter als ein iterativer Algorithmus zu finden. Iterative Algorithmen versuchen mit Hilfe von Schleifen und Wiederholungen das Problem zu lösen. Ein einfaches Beispiel ist das Suchen nach Dateien auf einem Rechner. Diese können in beliebig tiefen Unterverzeichnissen versteckt sein. Ohne eine genaue Kenntnis der Verzeichnisstrukturen lässt sich sehr einfach ein rekursives Verfahren aufstellen, vgl. Bsp. 2.7: Man durchsucht zunächst das aktuelle Verzeichnis. Im Anschluss wiederholt man die Suche in den Unterverzeichnissen. Aufgrund der Rekursion ist sichergestellt, dass alle Unterverzeichnisse durchsucht werden.

Beispiel 2.7: *Rekursiver Algorithmus zur Suche nach Dateien*

```
SucheDatei mit Name

Für alle Dateien im aktuellen Verzeichnis wiederhole:
     Falls Dateiname = Name:
          Ausgabe: Datei gefunden
          Terminiere

Für alle Unterverzeichnisse im aktuellen Verzeichnis
wiederhole:
     Wechsel in das Unterverzeichnis
     SucheDatei mit Name
     Wechsel in das alte Verzeichnis zurück
```

Weitere Klassen von Algorithmen erfüllen nicht alle oben genannten Kriterien: Beispielsweise enthalten Betriebssysteme Algorithmen, welche nicht-terminiert sind, da sie ihre Dienste fortwährend anbieten (z. B. Grafiktreiber). Selbstverständlich sollten auch diese Algorithmen zu einer gegebenen Eingabe nach endlicher Zeit eine Ausgabe erzeugen. Man kann sie daher im Sinne des EVA-Prinzips als terminiert ansehen, auch wenn die Ausführung endlos fortgesetzt wird.

Neben deterministischen Algorithmen gibt es zur Lösung spezieller Probleme eine Reihe von nicht-deterministischen Algorithmen, wie z. B. genetische Algorithmen oder stochastische Algorithmen. Diese Algorithmen sind in der Regel auch nicht-determiniert: der zweimalige Aufruf führt zu unterschiedlichen Ergebnissen[13].

Beispiel 2.8: *Autorouter: nicht-deterministischer Algorithmus*

Das automatische Routen[14] von Leiterbahnen erfolgt zunächst nach einfachen Kriterien wie die kürzeste Verbindung, die minimale Anzahl von Durchkontaktierungen etc. Mit wachsender Anzahl von Verbindungen kann jedoch keine Optimierung, z. B. durch das Testen aller Möglichkeiten, mehr erfolgen. Daher versucht der Router eine Optimierung durch das zufällige Entfernen bereits gerou-

13. Deterministische Algorithmen sind zwangsläufig determiniert, da auf gleichen Eingaben immer exakt die gleichen Arbeitsschritte und damit die gleichen Ergebnisse folgen. Nicht-deterministische Algorithmen können bei gleichen Eingaben gleiche Ausgaben liefern, z. B. wenn es zur Lösung eines Problems immer eine eindeutige Lösung gibt. In solchen Fällen ist das Ergebnis ebenfalls determiniert.

14. Routen: Erstellen von Verbindungen in einem Schaltungslayout.

teter Leiterbahnen zu finden. Da „zufällig“[15] hier Entscheidungen getroffen werden, können sich zwei Durchgänge erheblich unterscheiden.

Korrektheit und Komplexität

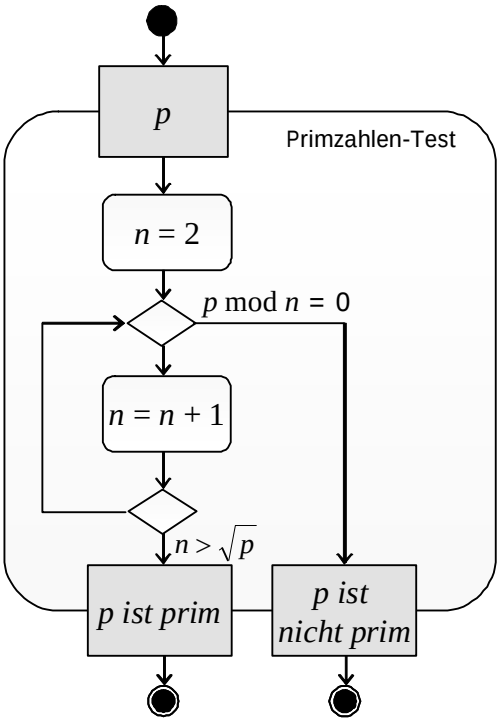
Es seien zuletzt noch die Eigenschaften *Korrektheit* und *Komplexität* von Algorithmen kurz angesprochen, welche ein weites Forschungsfeld in der theoretischen Informatik darstellen. Das höchste Ziel ist es, die Komplexität eines Algorithmus gering zu halten und gleichzeitig die Korrektheit zu gewähren. Eine Vielzahl von Problemklassen gilt jedoch aufgrund ihrer Komplexität als nicht lösbar, so dass eine geringe Komplexität bei Nachweis der Korrektheit oft als unerreichbar scheint.

Ein interessantes Beispiel stellt dazu die Überprüfung großer Zahlen auf ihre Eigenschaft als Primzahl dar[16].

Beispiel 2.9: *Deterministischer Algorithmus zur Überprüfung von Primzahlen*

```
Prüfe p auf Primzahl:

Wiederhole für n=2 bis n=Wurzel(p):

     falls Division p/n ohne Rest:
          terminiere mit Ausgabe: p ist keine Primzahl

Ausgabe: p ist Primzahl
```

Bild 2.11: *Primzahlen-Test*



15. Zufällig wird hier in Anführungszeichen gestellt, weil mit klassischen Computersystemen keine echtzufälligen Ereignisse generiert werden können. Vielmehr kommen sog. Pseudo-Zufallsgeneratoren zum Einsatz, welche auf mathematischen Zahlenfolgen, z. B. rückgekoppelten Schieberegistern basieren. Als Startwert dieser Folgen wird ein scheinbar zufälliger Wert, z. B. die aktuelle Uhrzeit etc. verwendet. Theoretisch können aber zwei identische Rechner über den exakt gleichen Startzustand verfügen, was zur Folge hätte, dass auf diesen Rechnern ein nicht deterministischer Algorithmus exakt gleich abläuft.

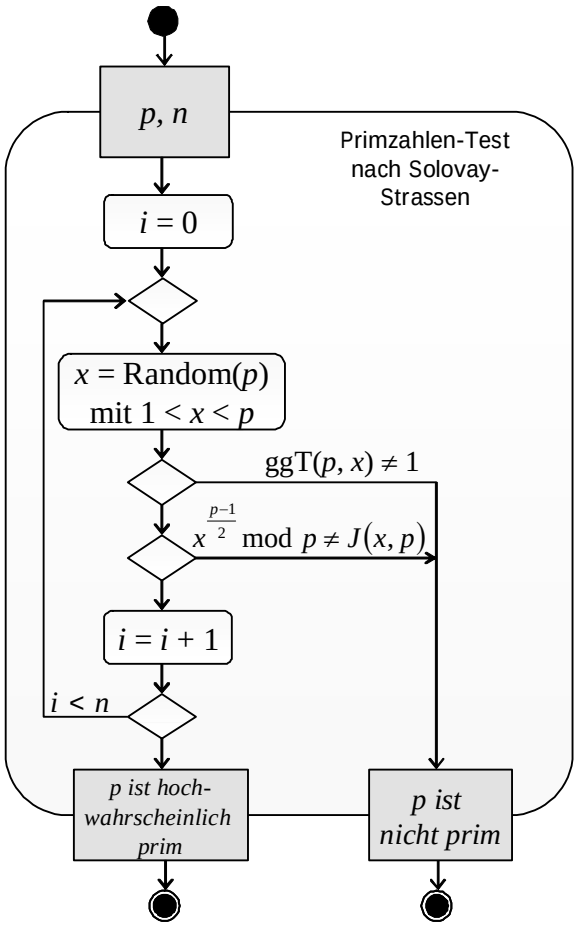
16. Große Primzahlen mit einigen hundert Dezimalstellen werden für die Verschlüsselung von Nachrichten verwendet: Ein Produkt zweier Primzahlen lässt sich für die Verschlüsselung leicht berechnen. Es ist aber sehr aufwendig, ein solches Produkt in seine beiden Primfaktoren zu zerlegen, was für eine Entschlüsselung (ohne Kenntnis des passenden Schlüssels) notwendig wäre, siehe: http://www.heise.de/newsticker/Weltrekord-im-Zahlenknacken--/meldung/59428.
Die derzeit (Stand 08/2008) größte bekannte und verifizierte Primzahl mit 9,8 Mio. Dezimalstellen ist $2^{32582657}$-1 (siehe http://www.mersenne.org/).

Mit dem Algorithmus nach Bsp. 2.9 würde eine 100-stellige Zahl 10^{50} Divisionen erfordern. Bei einer Rechenzeit von 1 µs pro Division würde es ca. $3 \cdot 10^{27}$ Milliarden Jahre dauern, bis eine Primzahl als solche identifiziert wird. Dafür prüft der Algorithmus vollständig auf die Eigenschaft prim und stellt damit einen korrekten Algorithmus für die Anforderung als Primzahlentest dar. Als Alternative bietet sich ein stochastischer Algorithmus an, der eine Zahl stichprobenartig auf ihre Eigenschaft *prim* überprüft. Es handelt sich um einen nicht-deterministischen Algorithmus:

Beispiel 2.10: *Primzahlentest nach Solovay-Strassen*

```
Prüfe p auf Primzahl:

Wiederhole für eine vorgegebene Anzahl n Stichproben:

     Erzeuge eine Zufallszahl x mit 1 < x < p

     Falls größter gemeinsamer Teiler GGT(p,x) ≠ 1:
          terminiere mit Ausgabe: p ist keine Primzahl

     Falls x^((p-1)/2) mod p ≠ J(x,p)
          terminiere mit Ausgabe: p ist keine Primzahl[17]

Ausgabe:
p ist Primzahl mit einer Fehlerwahrscheinlichkeit < 0.5^n
```

Bild 2.12: *Nicht-deterministischer Primzahlen-Test*



[17] Die linke Seite der Gleichung entspricht einer Variante des kleinen Fermatschen Satzes. Eine Primzahl oder Pseudoprimzahl liefert als Ergebnis entweder -1 oder 1. Auf der rechte Seite steht das Jacobi-Symbol $J(x,p)$.

Auch der Algorithmus nach Bsp. 2.10 ist korrekt, da er eine Umsetzung des Solovay-Strassen-Tests darstellt. Allerdings ist die Aussage „*p* ist eine Primzahl“ nur mit einer gewissen Wahrscheinlichkeit korrekt. Für große *n* ergibt sich eine sehr hohe Wahrscheinlichkeit. Dagegen ist die Aussage „*p* ist keine Primzahl“ immer korrekt.

Der Kernunterschied der beiden Algorithmen ist, dass der zweite Algorithmus nach Bsp. 2.10 in sehr kurzer Zeit auch für sehr große Zahlen terminiert und damit in der Praxis anwendbar ist.

Sicherlich muss in solchen Fällen zwischen hoher Effizienz (geringer Komplexität) und Korrektheit abgewägt werden. Generell sollte jedoch die Korrektheit das oberste Ziel bei der Entwicklung von Algorithmen sein. Da „durch Testen nur Fehler aufgedeckt werden, aber nicht die Abwesenheit von Fehlern nachgewiesen werden kann“ (E. W. Dijkstra), sollte die Korrektheit bereits in der Entwurfsphase von Algorithmen untersucht werden.

Ordnung

Für die Komplexität von Algorithmen findet man häufig eine Klassifizierung nach der Ordnung, welche mit $O(n)$ = *Ordnung von n* bezeichnet wird. Das *O* steht dabei für eine obere Schranke, d.h., auf einen Algorithmus abgebildet, werden für *n* Elemente (Datensätze) maximal $O(n)$ Operationen auf den Datensatz angewandt. Dabei kann *O(n)* vereinfacht als eine Funktion mit dem Eingabeparameter *n* betrachtet werden. Da nur der am stärksten wachsende Anteil betrachtet wird, schreibt man nur diesen und vernachlässigt die niedrigeren Anteile. Einige wichtige Klassen von Algorithmen sind im Folgenden aufgezählt:

- O(1) – *konstante Ordnung*: Der Algorithmus benötigt eine konstante Bearbeitungszeit unabhängig von der Menge der zu verarbeitenden Daten.
- $O(\mathrm{ld}\ n)$ – *logarithmische Ordnung*: Die Anzahl der Operationen verhält sich logarithmisch zu der Menge der Daten *n*, z. B. durch eine kontinuierliche Halbierung (Ausschließen) von Daten.
- $O(n)$ – *lineare Ordnung*: Ein Algorithmus, der auf jede der *n* Daten mit linearer Häufigkeit zugreift, also z. B. einmal, zweimal, zehnmal etc.
- $O(n^2)$ – *quadratische Ordnung*: Die Daten werden mit quadratischer Häufigkeit verarbeitet. Zum Beispiel beim paarweisen Vergleich aller *n* Elemente (also $n \times n$ Operationen).
- $O(n^3)$ – *kubische Ordnung*: Es folgt eine kubische Verarbeitung der Daten, wie z. B. die Multiplikation zweier $n \times n$ Matrizen.
- $O(2^n)$ – *exponentielle Ordnung*: Die Verarbeitung erfordert einen exponentiellen Zugriff auf die Daten, wie z. B. beim Erzeugen von vollständigen booleschen Wahrheitstabellen.
- $O(n!)$ – *faktorielle Ordnung*: Der Zugriff erfolgt mit der Fakultät von *n*, wie z. B. beim Aufzählen aller Permutationen von *n* Elementen.

O(1)	**$O(\mathrm{ld}\ n)$**	**$O(n)$**	**$O(n^2)$**	**$O(n^3)$**	**$O(2^n)$**	**$O(n!)$**
1	~ 3	10	100	1000	1024	3628800
1	~ 7	100	10000	$1 \cdot 10^6$	$\sim 1.27 \cdot 10^{30}$	$\sim 9.33 \cdot 10^{157}$
1	~ 10	1000	$1 \cdot 10^6$	$1 \cdot 10^9$	$\sim 1.1 \cdot 10^{301}$	$\sim 4.02 \cdot 10^{2567}$
1	~ 13	10000	$1 \cdot 10^8$	$1 \cdot 10^{12}$	$\sim 1.99 \cdot 10^{3010}$	$\sim 2{,}84 \cdot 10^{35659}$

Tabelle 2.4: *Wachstum der Operationen bei Algorithmen mit unterschiedlicher Ordnung*

Polynomielle Ordnung

Sofern sich die Anzahl der Operationen durch ein Polynom von *n* darstellen lässt, spricht man von einer *polynomiellen Ordnung*.

P-Probleme

Algorithmen mit polynomieller Ordnung gelten als handhabbar (P-Probleme), während Algorithmen mit exponentieller oder faktorieller Ordnung nur für sehr kleine *n* in endlicher Zeit terminieren. Dies zeigt Tabelle 2.4 ganz deutlich: Würde man die Anzahl der Operationen aus Tabelle 2.4 in einer Grafik über *n* darstellen, so würden die polynomiellen Ordnungen (linke Seite der Tabelle bis zum Doppelstrich) bei dem anzusetzenden Maßstab nur als eine Gerade auf der x-Achse erscheinen und das immense Wachstum der exponentiellen und faktoriellen Ordnungen würde die Grafik beherrschen. Man spricht daher von NP-Problemen. NP-Probleme sind mit einem deterministischen Algorithmus in polynomieller Zeit nicht lösbar. Wir benötigen einen nicht-deterministischen Algorithmus zur Lösung. In der Regel kann das Ergebnis aber in polynomieller Zeit auf Korrektheit überprüft werden. Beispiele für NP-Probleme sind:

NP-Probleme

◊ das Problem des Handlungsreisenden (*n* Städte mit Verbindungsstraßen, welche alle auf kürzestem Wege genau einmal zu bereisen sind),

◊ das Stundenplanproblem (lückenloser Stundenplan für alle Lehrer und Schüler).

Durch Stephen Cook (1971) wurde der Beweis erbracht: Wenn sich *ein* polynomieller Algorithmus für ein besonders komplexes NP-Problem, das sogenannte NP-vollständige Problem, finden lässt, könnten alle NP-Probleme polynomiell gelöst werden. Allerdings ist bis heute kein polynomieller Algorithmus zur Lösung eines NP-Problems gefunden worden[18].

Die Abgrenzung des Begriffes Algorithmus

Es gilt, zwischen einer Reihe von Begriffen rund um den Begriff Algorithmus in der Informatik detaillierter zu unterscheiden:

◊ Ein *Algorithmus* ist ein systematisches, reproduzierbares Problemlösungsverfahren, d.h., eine präzise, endliche Verarbeitungsvorschrift für eine Maschine oder einen Menschen, die oder der in der Lage ist, die im Algorithmus angegebenen Elementaroperationen schrittweise durchzuführen. Ein Algorithmus ist allgemein gültig und nicht auf eine spezielle Maschine zugeschnitten.

◊ Ein *Pattern* (auch *Template* genannt) ist das Rohgerüst eines Algorithmus, welches zur Lösung einer Vielzahl ähnlicher Problemfelder dient. Das *Pattern* (*engl.* für Muster) gibt vor, an welchen Stellen ein Algorithmus für die Umsetzung eines speziellen Problems weiter ausgeführt werden muss.

◊ *Programmierung* ist die Entwicklung von Programmen: Vom Entwurf der Anforderungen, Klärung der notwendigen Algorithmen, über die Implementierung in eine Programmiersprache bis hin zum Test des entwickelten Programms.

◊ Eine *Programmiersprache* ist eine formale Sprache zur Erstellung von Verarbeitungsanweisungen für Rechnersysteme. Die Programmiersprache erlaubt die Abbildung eines Algorithmus in eine maschineninterpretierbare Form.

◊ Ein *Programm* ist das Ergebnis der Programmierung, also die konkrete Implementierung von Algorithmen in einer Programmiersprache.

◊ Die *Implementierung* oder *Codierung* ist die Formulierung eines Algorithmus in einer Programmiersprache.

---

18. Sollte jemand einen genialen Einfall haben, wie ein polynomieller Algorithmus zur Lösung einer NP-vollständigen Aufgabe aussehen könnte, so erwartet ihn unter anderem der mit US $ 1.000.000 dotierte Preis zur Lösung des *NP vs. P – Millenium Problems* vom Clay Mathematics Institute in Cambridge, Massachusetts: http://www.claymath.org/millennium/P_vs_NP/

◊ Unter einem *Prozess* versteht man den Ablauf und die Ausführung eines Programms.

◊ Der *Prozessor* ist die ausführende Instanz eines Prozesses.

Literatur

[Abel01] Harold Abelson, Gerald Jay Sussman: *Struktur und Interpretation von Computerprogrammen: eine Informatik-Einführung*. Berlin [u.a.]: Springer 2001 (4. Aufl.)

[Bro91] I. N. Bronstein, K. A. Semendjajew: *Taschenbuch der Mathematik*. Stuttgart [u.a.]: B. G. Teubner Verlagsgesellschaft und Verlag Nauka Moskau, 1991

[Hero07] Helmut Herold, Bruno Lurz, Jürgen Wohlrab: *Grundlagen der Informatik: praktisch - technisch - theoretisch*. München [u.a.]: Pearson Studium 2007, Kap. 2, 4-5

[Pom08] G. Pomberger, H. Dobler: *Algorithmen und Datenstrukturen - Eine systematische Einführung in die Programmierung*. München [u.a.]: Pearson Studium 2008, Kap. 1-2

[Sedg02] Robert Sedgewick: *Algorithmen*. München [u.a.]: Pearson Studium 2002 (2. Aufl.)

[Swo99] J. Swoboda: *Skript zur Vorlesung Grundlagen der Informatik*. München: TUM, Fak. Elektrotechnik und Informationstechnik, WS 1999

[Vogt04] Carsten Vogt: *Informatik: eine Einführung in Theorie und Praxis*. Heidelberg [u.a.]: Spektrum Akad. Verl. 2004

2.2 Die Datendarstellung im Rechner

Beim Entwurf von Algorithmen ist die Darstellung der Daten in einem Rechnersystem zunächst uninteressant; man kann von einem idealen System ausgehen. Für die Implementierung eines Algorithmus ist die Datendarstellung jedoch entscheidend, da sie sowohl auf die Vollständigkeit, Korrektheit als auch auf die Terminiertheit Einfluss nehmen kann. Für die korrekte Umsetzung eines Algorithmus ist daher ein tiefgehendes Verständnis der Datendarstellung im Rechner erforderlich. So kann der in Bsp. 2.5 gezeigte Algorithmus zur Berechnung der Fakultät mit einer 32-Bit-Kodierung maximal für Werte von $n = 12$ angewendet werden, wie sich über den *Logarithmus Dualis* leicht nachweisen lässt:

Beispiel 2.11: *Grenzen bei der Berechnung der Fakultät*

$$ld(12!) = 28.835... \approx 29 < 32$$

$$ld(13!) = 32.536... \approx 33 > 32$$