Algorithmen und Datenstrukturen

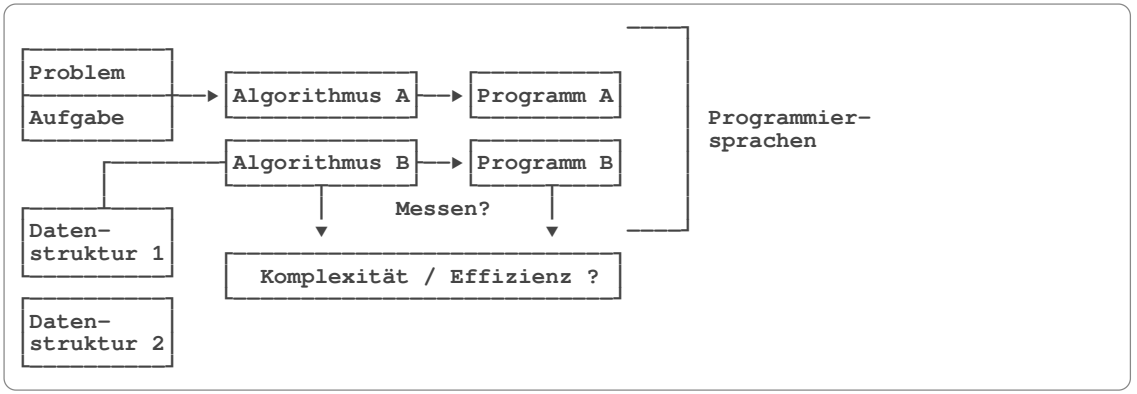
Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Einführung in Algorithmen und Datenstrukturen

1. Algorithmen
2. Datenstrukturen
3. Laufzeit und Komplexität



Literatur

Algorithmen und Datenstrukturen
Thomas Ottman
Peter Widmayer
Spektrum Verlag

Algorithmen und Datenstrukturen
Eine Einführung in Java
Gunter Saake, Kai Uwe Sattler
dpunkt Verlag

Historischer Überblick: Algorithmen

300 v. Chr.

Euklids Algorithmus zur Bestimmung des ggT (beschrieben im 7. Buch der Elemente), also des größten gemeinsamen Teilers, mit dem etwa ggT(300, 200) = 100 sehr effizient berechnet werden kann, ist die erste Beschreibung eines Verfahrens, das modernen Kriterien für Algorithmen gerecht wird.

800 n. Chr.

Der persisch-arabische Mathematiker Muhammed ibn Musa abu Djafar alChoresmi (oft auch als »al Chworesmi« oder »al Charismi« geschrieben) veröffentlicht eine Aufgabensammlung für Kaufleute und Testamentsvollstrecker, die später ins Lateinische als Liber Algorithmi übersetzt wird.

Das Wort Algorithmus ist ein Kunstwort aus dem Namen dieses Mathematikers und dem griechischen »arithmos« für Zahl.

1574

Adam Rieses Rechenbuch verbreitet mathematische Algorithmen in Deutschland.

1614

Die ersten Logarithmentafeln werden algorithmisch berechnet – ohne Computer dauerte dies 30 Jahre!

1703

Binäre Zahlensysteme werden von Leibnitz eingeführt. Diese Zahlensysteme bilden die Grundlage der internen Verarbeitung in modernen Computern.

Historischer Überblick: Algorithmen

1815

Augusta Ada Lovelace, die erste »Computer-Pionierin«, wird geboren. Sie entwickelt schon früh Konstruktionspläne für verschiedenartige Maschinen, wird Assistentin von Babbage und entwirft Programme für dessen erste Rechenmaschinen.

1822

Charles Babbage entwickelt die sogenannte Difference Engine, die in einer verbesserten Version 1833 fertiggestellt wird. Später entwickelt er auch die Analytical Engine, die bereits die wichtigsten Komponenten eines Computers umfasst, aber niemals vollendet wird.

1931

Gödels Unvollständigkeitssatz beendet den Traum vieler damaliger Mathematiker, die gesamte Beweisführung aller Sätze in der Mathematik mit algorithmisch konstruierten Beweisen durchzuführen.

1936

Die Church'sche These vereinheitlicht die Welt der Sprachen zur Notation von Algorithmen, indem für viele der damaligen Notationen die gleiche Ausdrucksfähigkeit postuliert wird.

danach

Mit der Realisierung der ersten Computer erfolgte natürlich der Ausbau der Algorithmentheorie zu einem eigenen Wissenschaftsgebiet.

Auch für den Bereich der Datenstrukturen können lang zurückreichende Wurzeln gefunden werden, etwa das Indexierungssystem historischer Bibliotheken.

Praktische Eigenschaften von Algorithmen

Bausteine für Algorithmen

1. Die Basiselemente sind die **elementaren Operationen**, die ausgeführt werden.
2. Das Hintereinanderausführen von Schritten bezeichnet man als **sequenzielle Ausführung**.
3. Die **parallele Ausführung** hingegen bedeutet das gleichzeitige Ausführen von Schritten.
4. Eine **bedingten Ausführung** ist ein Schritt, der nur ausgeführt wird, wenn eine bestimmte Bedingung erfüllt wird.
5. Die **Schleife** ist eine Wiederholung von Anweisungen, bis eine vorgegebene Endbedingung erfüllt wird ⇒ *Terminierungskriterium*!
6. Konzept des Unterprogramms: **Funktionen** und **Funktionsrekursion**.
7. Daten! Rekursion gibt es bei Funktionen und Daten!

Praktische Eigenschaften von Algorithmen

Bausteine für Algorithmen

1. Die Basiselemente sind die **elementaren Operationen**, die ausgeführt werden.
2. Das Hintereinanderausführen von Schritten bezeichnet man als **sequenzielle Ausführung**.
3. Die **parallele Ausführung** hingegen bedeutet das gleichzeitige Ausführen von Schritten.
4. Eine **bedingten Ausführung** ist ein Schritt, der nur ausgeführt wird, wenn eine bestimmte Bedingung erfüllt wird.
5. Die **Schleife** ist eine Wiederholung von Anweisungen, bis eine vorgegebene Endbedingung erfüllt wird ⇒ *Terminierungskriterium*!
6. Konzept des Unterprogramms: **Funktionen** und **Funktionsrekursion**.
7. Daten! Rekursion gibt es bei Funktionen und Daten!

Algorithmen verarbeiten Daten. Daten müssen wie Algorithmen strukturiert werden. Daher der Name dieser Veranstaltung.

Formale Beschreibung von Algorithmen

Ein Algorithmus beschreibt eine Abbildung:

$$f : E \rightarrow A$$

mit

E

Menge der zulässigen Eingabedaten.

A

Menge der Ausgabedaten.

Aber: Nicht jede Abbildung f: $E \rightarrow A$ lässt sich durch einen Algorithmus realisieren (Berechenbarkeit!)

Formale Eigenschaften von Algorithmen

Welche formalen Eigenschaften von Algorithmen werden studiert?

Formale Eigenschaften von Algorithmen

Welche formalen Eigenschaften von Algorithmen werden studiert?

Die wichtigste formale Eigenschaft eines Algorithmus ist zweifellos dessen Korrektheit!

- Dazu muss gezeigt werden, dass der Algorithmus die jeweils gestellte Aufgabe richtig löst.

Formale Eigenschaften von Algorithmen

Welche formalen Eigenschaften von Algorithmen werden studiert?

Die wichtigste formale Eigenschaft eines Algorithmus ist zweifellos dessen Korrektheit!

- Dazu muss gezeigt werden, dass der Algorithmus die jeweils gestellte Aufgabe richtig löst.

Aber kann die Korrektheit (d.h. Fehlerfreiheit) eines Algorithmus oder besser dessen Implementierung mit einer Programmiersprache überhaupt bewiesen werden? Uff.

Formale Eigenschaften von Algorithmen

Die zweite wichtige formale Eigenschaft eines Algorithmus ist dessen Effizienz.

- Da sieht es hier schon besser aus. **Effizienz ist messbar, Korrektheit nicht.**

Wir müssen zwischen Algorithmen (also Berechungsvorschriften) und deren Implementierung mit einer Programmiersprache unterscheiden. Welche Programmiersprachen und Programmiermodelle?

Programmiersprachen und Programmiermodelle

Algorithmen werden mit Programmiersprachen implementiert (Handwerkzeug)

Es gibt verschiedene Programmiermodelle und Klassen:

1. **Funktionaler Programmierung** (mathematisch, zustandslos, z.B. Haskell, ML, LISP, OCAML)
2. **Prozedurale und imperative Programmierung** (C, ggfs. JavaScript, Fortran)
3. **Objektorientierte Programmierung** (Java, ggf, JavaScript und Python)
4. **Deklarative (deduktive) und logische Programmierung** (Prolog)

Laufzeit und Komplexität

Die Laufzeit ist messbar, die Rechenkomplexität nur bestimmbar/ableitbar!

- Annahme: Einheitsoperations (Addition, Wertzuweisung usw.)
- Rechenkomplexität ist spezifisch für einen Algorithmus, nicht für eine Programmierung (da Laufzeit)

Rechnekomplexität bedingt auch eine Speicherkomplexität!

- Speicherkomplexität meist einfacher abzuschätzen; aber
  - Nur statischer Datenspeicher lässt sich direkt aus den Ein- und Augabedaten ableiten
  - Dynamischer Speicherbedarf z.B. bei Funktionsaufrufen ist gesondert und dezidiert zu berechnen

Laufzeit und Komplexität

Beispiel statischer und dynamischer Speichebedarf

```
D=[1,2,3,4,5,6,7,8,9,10] // N=10, |A|=10*B
sum=0
// Fall 1
for(i=0;i<D.length;i++) sum+=D[i]
// Fall 2
function dosum(i) {
  if (i>0) D[i]+dosum(i-1);
  return D[0]
}
sum=dosum(D.length-1)
```

Wo liegen die Unterschiede in Fall 1 und 2 beim Speicherbedarf bezogen auf *N*?

Laufzeit und Komplexität

Fall 1

- Rechenzeit (in Einheitsoperationen) ist $N*k$
- Speicherbedarf (in Einheitszellen) ist $N+k$

Fall 2

- Rechenzeit (in Einheitsoperationen) ist N*k
- Speicherbedarf (in Einheitszellen) ist $N+N*k$

Messbarkeit von Effizienz

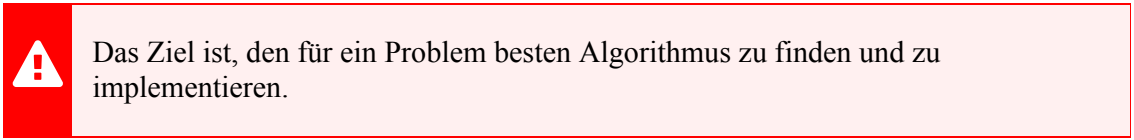
- Nochmal wichtigste Eiegenschaften eines Algorithmus: Korrektheit und Effizienz
- Man könnte beides durch Implementation des Algorithmus in einer konkreten Programmiersprache auf einem konkreten Rechner für eine Menge repräsentativ gewählter Eingaben messen.

Solche experimentell ermittelten Meßergebnisse lassen sich aber nicht oder nur schwer auf andere Implementationen und andere Rechner übertragen und **sind nicht vollständig, sondern immer exemplarisch**.

- Nicht Test (oder Simulation) mit Nachweis von Korrektheit verwechseln! Das gibt nur Frust bei den Kunden und endlose Rückrufaktionen....

Messbarkeit von Effizienz

- Man kann häufig dasselbe Problem durchaus mit verschiedenen Algorithmen lösen.



- Das verlangt insbesondere eine möglichst optimale Nutzung der Ressourcen Speicherplatz und Rechenzeit.

Messbarkeit von Effizienz

Beispiel: Ein von Jon Bentley behandeltes Problem

> Gegeben sei eine Folge X von N ganzen Zahlen in einem Array. Gesucht ist die maximale Summe aller Elemente in einer zusammenhängenden Teilfolge. Sie wird als maximale Teilsumme bezeichnet, jede solche Folge als maximale Teilfolge.

1	2	3	4	5	6	7	8	9	10	Index
31	-41	59	26	-53	58	97	-93	-23	84	Zahlen

Bsp. 1. Für die Eingabefolge X[1::10] ist die Summe der Teilfolge X[3::7] mit Wert 187 die Lösung des Problems.

Naives Verfahren (Vollständige Schleifeniteration)

```
maxtsumme = 0
for (u=0;u<N;u++) {
  for (o=u;o<N;o++) {
    // bestimme die Summe der Elemente in der Teilfolge X [ u : : o ]
    Summe = 0;
    for (i=u; i<=o;i++) Summe = Summe + X[i];
    // bestimme den größeren der beiden Werte Summe und maxtsumme
    maxtsumme = max (Summe, maxtsumme)
  }
}
```

Alg. 1. Naives Verfahren für die Suche nach der größten Teilsumme

Anzahl der Einheitsoperationen:

$$\sum_{u=0}^{N-1}\sum_{o=u}^{N-1}\sum_{i=u}^{o}1 = N(N-1+N-2+..+1)(1+2+3+..+N-1) = f\left(N^3\right)$$

teilsumme1.js

```
X=[31,-41,59,26,-53,58,97,-93,-23 ,84 ]
maxtsumme = 0
N=X.length
for (u=0;u<N;u++) {
  for (o=u;o<N;o++) {
    // bestimme die Summe der Elemente in der Teilfolge X [ u : : o ]
    Summe = 0;
    for (i=u; i<=o;i++) Summe = Summe + X[i];
    // bestimme den größeren der beiden Werte Summe und maxtsumme
    maxtsumme = max (Summe, maxtsumme)
  }
}
print(maxtsumme)
```

Divide-and-Conquer (Zerlegung)

Jetzt folgen wir dem Divide-and-conquer-Prinzip (DaC) zur Lösung des Maximum-Subarray-Problems. Die Anwendbarkeit dieses Prinzips ergibt sich aus folgender Überlegung:

- Wird eine gegebene Folge in der Mitte geteilt, so liegt die maximale Teilfolge entweder
  - ganz in einem der beiden Teile
  - oder sie umfaßt die Trennstelle, liegt also teils im linken und teils im rechten Teil.
- Im letzteren Fall gilt für das in einem Teil liegende Stück der maximalen Teilfolge:
  - Die Summe der Elemente ist maximal unter allen zusammenhängenden Teilfolgen in diesem Teil, die das Randelement an der Trennstelle enthalten.

Warum Divide-and-Conquer?

Divide-and-Conquer (Zerlegung)

Jetzt folgen wir dem Divide-and-conquer-Prinzip (DaC) zur Lösung des Maximum-Subarray-Problems. Die Anwendbarkeit dieses Prinzips ergibt sich aus folgender Überlegung:

- Wird eine gegebene Folge in der Mitte geteilt, so liegt die maximale Teilfolge entweder
  - ganz in einem der beiden Teile
  - oder sie umfaßt die Trennstelle, liegt also teils im linken und teils im rechten Teil.
- Im letzteren Fall gilt für das in einem Teil liegende Stück der maximalen Teilfolge:
  - Die Summe der Elemente ist maximal unter allen zusammenhängenden Teilfolgen in diesem Teil, die das Randelement an der Trennstelle enthalten.

Warum Divide-and-Conquer?

DaC basiert auf Rekursion und rekursive Zerlegung eines Problems in kleine (elementare) Probleme.

- Wir wollen die maximale Summe von Elementen, die das linke bzw. das rechte Randelement einer Folge von Elementen enthält, kurz das linke bzw. rechte Randmaximum nennen.
- Das linke Randmaximum $l_{\max}$ für eine Folge X[l] ... X[r] ganzer Zahlen kann man in ungefähr (r-l) Schritten (also linear) wie folgt bestimmen:

```
lmax = 0
summe = 0
for (i=l;i<=r;i++) {
  summe = summe + X[i]
  lmax = max (lmax,summe)
}
```

Alg. 2.

- Entsprechend kann man auch das rechte Randmaximum $r_{\max}$ für eine Folge ganzer Zahlen in einer Anzahl von Schritten bestimmen, die linear mit der Anzahl der Folgenelemente wächst.

```
function maxtsum (X) {
 // f liefert eine maximale Teilsumme der Folge X ganzer Zahlen g
  if (length(X)==1) {
    a=X[0]
    if (a > 0) result = a
    else result = 0
  } else {
   // Divide:
   // teile X in eine linke und eine rechte Teilfolge A und B annähernd gleicher Größe;
   j = floor(length(X)/2)
   A = slice(X,[0,j]); B=slice(X,[j+1,length(X)-1])
   // Conquer:
   maxtinA = maxtsum (A)
   maxtinB = maxtsum (B)
   // bestimme das rechte Randmaximum rmax ( A ) der linken Teilfolge A;
   // bestimme das linke Randmaximum lmax ( B ) der  rechten Teilfolge B;
   // Merge:
   result = max(maxtinA, maxtinB, rmax(A) + lmax (B))
  }
  return result
}
```

Alg. 3. Der gesamte DaC Algorithmus zur Suche der maximalen Teilsumme

Bezeichnet nun *T*(*N*) die Anzahl der Schritte, die erforderlich ist, um den Algorithmus *maxtsum* für eine Folge der Länge *N* auszuführen, so gilt offenbar folgende Rekursionsformel:

$$T(N) = 2T\left(\frac{N}{2}\right) + Const \cdot N$$

Man erhält dann für die Größenordnung der Berechungen (also *maxtsum* Aufrufe) folgende asymptotische Näherung:

$$T(N) = f(N \cdot \log(N))$$

- Das ist schon viel besser als die Laufzeit des naiven Verfahrens.
- Aber ist es bereits das bestmögliche Verfahren? Nein — denn die Anwendung eines weiteren algorithmischen Lösungsprinzips, des Scan-line-Prinzips, liefert uns ein noch besseres Verfahren.

Scan-line Verfahren (Abtastung)

- Wir haben eine aufsteigend sortierte, lineare Folge von Inspektionsstellen (oder: Ereignispunkten), die Positionen 1...*N* der Eingabefolge.
- Wir durchlaufen die Eingabe in der durch die Inspektionsstellen vorgegebenen Reihenfolge
  - Dabei führen wir zugleich eine vom jeweiligen Problem abhängige Information mit,
  - eine dynamisch veränderliche, d.h. an jeder Inspektionsstelle gegebenenfalls zu korrigierende Information.
  - Hier: die maximale Summe *bisMax* einer Teilfolge im gesamten bisher inspizierten Anfangsstück und das an der Inspektionsstelle endende rechte Randmaximum *ScanMax* des bisher inspizierten Anfangsstücks

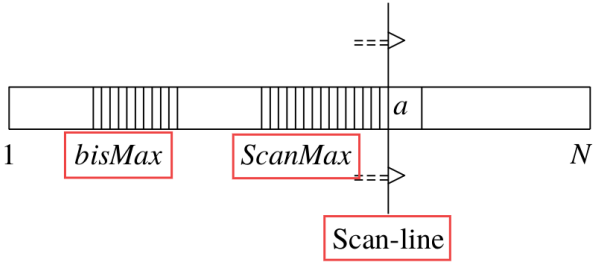


Abb. 1. Prinzip des aufsteigenden Scan-line Verfahrens mit den bisMax und ScanMax Variablen

- Nehmen wir nun an, daß wir bereits ein Anfangsstück der Länge *l* der gegebenen Folge inspiziert haben und die maximale Teilsumme *bisMax* sowie das rechte Randmaximum *ScanMax* in diesem Anfangsstück kennen.
- Was ist die maximale Teilsumme, wenn man das (*l*+1)-te Element, sagen wir *a*, hinzunimmt?
  - Die maximale Teilfolge des neuen Anfangsstücks der Länge $l + 1$ liegt entweder bereits im Anfangsstück der Länge *l*,
  - oder sie enthält das neu hinzugenommene Element *a*, reicht also bis zum rechten Rand.

- Das rechte Randmaximum der neuen Folge mit $l + 1$ Elementen erhält man nun aus dem rechten Randmaximum der Folge durch Hinzunahme von a, also $ScanMax + a$, vorausgesetzt, daß dieser Wert insgesamt positiv bleibt.
  - Ist das nicht der Fall, so ist die maximale Summe von Elementen, die das rechte Randelement enthält, die Summe der Elemente der leeren Folge, also 0.

```
Q = [1,2,..,N]
// Folge der Inspektionsstellen von links nach rechts = Folge der Positionen 1 ... N
// Initialisiere
ScanMax = 0;
bisMax = 0;
while (length(Q)) {
  q = head(Q); Q=tail(Q)
  a = X[Q[q]]
  // update ScanMax und bisMax
  if ((ScanMax + a) > 0) ScanMax = ScanMax + a
  else ScanMax = 0;
  bisMax = max ( bisMax, ScanMax )
}
```

Alg. 4. Vollständiger Algorithmus des Scan-line Verfahrens für die Maximumsuche. Am Ende enthält dann bisMax den gewünschten Wert.

- Das Scan-line Verfahren arbeitet in linearer Zeit!

$$T(N) = f(N)$$

- Solch einen Forschritt in der Steigerung der Berechnungseffizienz gibt es aber nicht immer.

Achtung: Wie betrechten bisher nur algorithmische Durchläufe von Berechnungen bie der Laufzeit. Tatsächlich muss man sich die Gesamtzahl an Rechneoperationen anschauen um eine Aussage der Effiziez und Vergleichbarkeit zu ermöglichen. Das ist vor allem bei "kleineren" N ein signifikantes Problem.

- Beispiel: Die Listen (Vektor) Funktionen *head* und *tail* die wir im Scan-line Algorithmus verwendet haben besitzen u.U. eine nicht konstante Laufzeit (von ihrem *n* ausgesehen).

Schon kleine Änderung in der Implementierung können signifikanten Einfluss auf die Gesamtrechenzeit habem (aber nicht unbedingt auf f(N)!)

```
Q = [1,2,..,N]
// Folge der Inspektionsstellen von links nach rechts = Folge der Positionen 1 ... N
// Initialisiere
ScanMax = 0;
bisMax = 0;
for (q=0;q<length(Q);q++) {
  a = X[Q[q]]
  // update ScanMax und bisMax
  if ((ScanMax + a) > 0) ScanMax = ScanMax + a
  else ScanMax = 0;
  bisMax = max ( bisMax, ScanMax )
}
```

Alg. 5. Version 2: Vollständiger Algorithmus des Scan-line Verfahrens für die Maximumsuche. Am Ende enthält dann bisMax den gewünschten Wert.

Beweis der Korrektheit

Beispiel: Multiplikation zweier Binärzahlen *a* und *b* so dass $z = a*b$ ist.

```
x = a; y = b;z = 0;
while (y > 0) {
 if (!odd(y)) {
  y = y / 2;
  x = x + x
 } else {
  y = y - 1
  z = z + x
 }
}
```

Beweis der Korrektheit

```
x         y
1 1 0 1 • 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
-------------------------
  1 0 0 0 0 0 1
```

x	y	z	Anzahl Schleifeniterationen
1101	101	0	0
1101	100	1101	1
11010	10	1101	2
110100	1	1101	3
110100	0	1000001	4

Code 1. Schrittweise Multiplikation durch Schieben und Addieren

- Es ist nicht schwer, die gleichen Rechenschritte wiederzuerkennen, die vorher beim aus der Schule bekannten Verfahren zur Multiplikation dieser zwei Zahlen ausgeführt wurden.
- Ein Beweis für die Korrektheit des Verfahrens ist diese Beobachtung jedoch nicht.

Beweis der Korrektheit

- Dazu müssen wir vielmehr zeigen, daß für zwei beliebige nichtnegative ganze Zahlen *a* und *b* gilt, daß das Programmstück für diese Zahlen terminiert und es das Produkt der Zahlen *a* und *b* als Wert der Variablen *z* liefert.

Um das zu zeigen, benutzen wir eine sogenannte Schleifeninvariante; das ist eine den Zustand der Rechnung charakterisierende, von den Variablen abhängende Bedingung. In unserem Fall nehmen wir die Bedingung:

$$P : y \geq 0 \wedge z + x \cdot y = a \cdot b$$

und zeigen, daß die folgenden drei Behauptungen gelten.

Beweis der Korrektheit

Behauptung 1

P ist vor Ausführung der while-Schleife richtig, d.h. vor erstmaliger Ausführung der if-Anweisung {*}.

Behauptung 2

P bleibt bei einmaliger Ausführung der in der while-Schleife zu iterierenden Anweisung richtig. D h. genauer, gelten die die while-Schleife kontrollierende Bedingung und die Bedingung *P* vor Ausführung der Anweisung {*}, so gilt nach Ausführung der Anweisung {*} ebenfalls *P*.

Behauptung 3

Die in der while-Schleife zu iterierende if-Anweisung wird nur endlich oft ausgeführt. Man sagt statt dessen auch kurz, daß die while-Schleife terminiert.

Nehmen wir einmal an, diese drei Behauptungen seien bereits bewiesen. Dann er-halten wir die gewünschte Aussage, daß das Programmstück terminiert und am Ende $z = a \bullet b$ ist, mit den folgenden Überlegungen.

Beweis der Korrektheit

- Die Gültigkeit von Behauptung 1 ist offensichtlich.
- Dass das Programmstück für beliebige Zahlen *a* und *b* terminiert, folgt sofort aus Behauptung 3.
- Wegen Behauptung 1 und Behauptung 2 muss nach der letzten Ausführung der in der while-Schleife zu iterierenden Anweisung {*} *P* gelten und die die while-Schleife kontrollierende Bedingung $y > 0$ natürlich falsch sein
- Zum Nachweis von Behauptung 2 nehmen wir an, es gelte vor Ausführung der if-Anweisung {*}:

$$(y \geq 0 \wedge z + xy = ab) \wedge (y > 0)$$

Fall 1: [y ist gerade] Dann wird y halbiert und x verdoppelt. Es gilt also nach Ausführung der if-Anweisung {*} immer noch (y ≥ 0 und z + x • y = a • b).

Fall 2: [y ist ungerade] Dann wird y um 1 verringert und z um x erhöht und daher gilt ebenfalls nach Ausführung der if-Anweisung wieder (y ≥ 0 und z + x • y = a • b).

Beweis der Korrektheit

- Zum Nachweis der Behauptung 3 genügt es zu bemerken, dass bei jeder Ausführung der in der while-Schleife zu iterierenden if-Anweisung der Wert von y um mindestens 1 abnimmt.
- Nach höchstens y Iterationen muß also $y \leq 0$ werden und damit die Schleife terminieren.
- Damit ist insgesamt die Korrektheit dieses Multiplikationsalgorithmus bewiesen.

Das war ein sehr einfacher Algorithmus (10 Zeilen Pseudocode). Der Linux Kernel besteht aus mehr als 10 Millionen Zeilen Programmcode. Viel Spaß beim Beweisen! Reale Programe können nicht bewiesen werden (was zu beweisen wäre).

□

Kosten von Algorithmen

Wie effizient ist das angegebene Multiplikationsverfahren?

- Zunächst ist klar, daß das Verfahren nur konstanten Speicherplatz benötigt, wenn man das **Einheitskostenmaß** zu-grundelegt, denn es werden nur drei Variablen zur Aufnahme beliebig großer ganzer Zahlen verwendet.
- Legt man das **logarithmische Kostenmaß** zugrunde, ist der Speicherplatzbedarf linear von der Summe der Längen der zu multiplizierenden Zahlen abhängig.
- Es macht wenig Sinn, zur Messung der Laufzeit das Einheitskostenmaß zugrunde zu legen. Denn in diesem Maß gemessen ist die Problemgröße konstant gleich 2.
- Wir interessieren uns daher für die Anzahl der ausgeführten arithmetischen Operationen in Abhängigkeit von der Länge und damit der Größe der zwei zu multiplizierenden Zahlen *a* und *b*.

□ *Gesamtlaufzeit von der Größenordnung O(Länge(b) • (Länge(a) + Länge(b)))*

Beschreibung von Algorithmen

1. Grafisch: Flussdiagramme
2. Grafisch: Blockdiagramme
3. Textuell: Pseudonotation ⇒ Künstliche nicht standardisierte Programmierung mit Text und mathematischer Notation
4. Textuell: Konkrete Programmiersprache

In diesem Kurs:

1. Pseudonotation ist JavaScript (nur pre-ES2015, Variablen und Funktionen) ⇒ Vorlesung
2. Programmiersprache ist Java ⇒ Übung ⇒ Transferleistung (Prozedural ⇒ OO)

Beschreibung von Algorithmen

Pseudo-Js

```
x={n:1}
function up(counter) {
  counter.n=counter.n+1
}
function down(counter) {
  counter.n=counter.n-1
}
up(x)
```

JAVA

```
public class Counter {
  int n;
  void up() {
    n=n+1
  }
  void down() {
    n=n-1
  }
}
Counter x=new Counter();
```

Bsp. 2. Ein Beispiel Pseudo-Js ⇔ JAVA

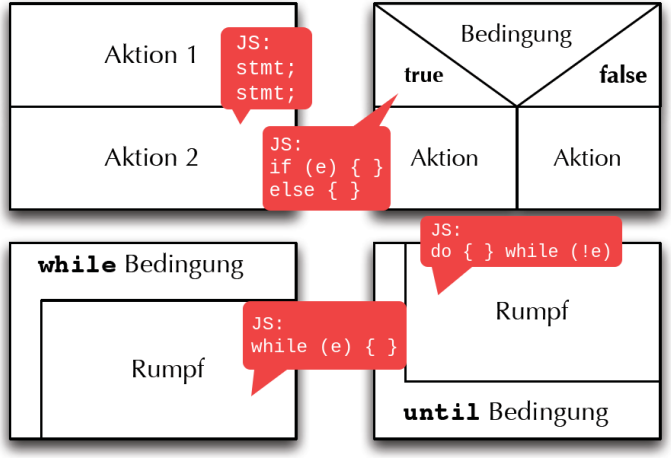
Pseudonotationen

Pseudocode-Notation für Algorithmen

- Die eingangs genannten Bausteine sind dafür geeignet, auf der intuitiven Ebene (also nicht auf der Basis mathematischer Strenge) Algorithmen zu formulieren.
- Ein Aspekt der Formulierung ist die Nutzung einer genormten, festgelegten Ausdrucksweise.
- Wir werden eine derartige Ausdrucksweise kennen lernen, die die genannten Bausteine benutzt und auf einer leicht verständlichen Ebene bleibt.
- Da die Formulierung der Algorithmen nahe an den Konstrukten verbreiteter Programmiersprachen ist, in denen Programme »kodiert« werden, bezeichnet man diese intuitiven Algorithmen auch als **Pseudocode-Algorithmen**.

Pseudonotationen

Struktogramme



[A&D Dpunkt Verlag]

Abb. 2. Notation für Struktogramme

Pseudonotation JavaScript

- Vorteil: Einfach zu verstehen, dynamische typisiert, d.h., kein Ballast von Typdeklaration, aber dennoch direkt z.B. im Browser ausführbar
- Bei Algorithmen und Datenstrukturen spielt Typisierung eine untergeordnete Rolle, erst bei Implementierung wichtig.
- Wir benutzen nur eine kleine Untermenge von JS, syntaktisch kompatibel zu Java, operational aber vereinfacht.
- Pseudonotation: rein prozedural, Java: rein Objektorientiert (mit einigen Ausnahmen)

Was wird benutzte:

1. Variablen x
2. Ausdrücke (arithmetisch, relational, logisch)
3. Kontrollanweisungen (Schleifen, bedingte Verzweigung)
4. Funktionen

That's all folks!

Pseudonotation JavaScript

Variablen und Ausdrücke

- Variablen werden on-the-fly ohne Deklaration eingeführt:

```
x = ε
// Arithmetisch/funktional
ε := a+b, a*b, a/b, a%b, a<<b, a>>b, -a, foo(a,b,c,..)
// Boolessch
ε := a&&v, a||b, !a
// Logisch
ε := a&v, a|b, ~a
```

- Jede variable ist nur eine Referenz auf Werte, also x=Ref(Value(ε)).

Funktionen

- Funktionen bestehen aus einem Namen (ggfs. als Lambdaausdruck namenlos), einer Parameterliste (Funktionsargumente, Variablen), und einem Funktionsrumpf (Anweisungen).
- Funktionen können direkt in Ausdrücken evaluiert (aufgerufen) werden und liefern Werte mittels der `return` Anweisung.

```
function foo(a,b,c,..) {
  ...
  return ε
}
x=foo(ε1,ε2,ε3,..)
```

Kontrollanweisungen

Bedingte Verzweigung

```
if (ε) {
  ..
}
if (ε) {
 ..
} else {
 ..
}
```

Mehrfachauswahl

```
switch (ε) {
  case c1:
    ..
    break;
  case c2:
    ..
    break;
  case c3:
    ..
    break;
  default :
    ..
}
```

Schleifen

```
for(i=ε;i<ε;i=ε(i)) {
  ..
}
while (ε) {
  ..
}
do {
  ..
} while (ε)
```

Schleifen

Man unterscheidet:

1. Zählschleife `for(start;condition;update)` mit drei integrierten Anweisungen: Einer Initialisierung (einer oder mehrere Zählvariablen), einer Schleifenbedingung die vor jedem Durchlauf getestet wird (i.A. der Zählvariable), und einer Update Anweisung die nach jedem Schleifendurchlauf ausgeführt wird (i.A. Veränderung der Zählvariable).

2. Abweisende Schleife `while(condition)` die abhängig von der Schleifenbedingung niemals, einmal oder mehrmals den Schleifenkörper ausführt (Test vor Schleifendurchlauf).

3. Eine nicht abweisende Schleife `do {} while(condfition)` die mindestens einmal den Schleifenkörper ausführt (Test nach jedem Schleifendurchlauf).

Konkrete versa Abstrakte Datentypen

1. Konkrete Datentypen werden von einer Programmiersprache direkt unterstützt, z.B. Arrays oder Rekords (Strukturen). Der Zugriff und die Nutzung dieser Datentypen benötigt keine nutzerdefinierten Funktionen.
2. Abstrakte Datentypen werden nicht direkt von der Programmiersprache unterstützt und müssen implementiert werden. Das bedeutet Implementierung (Definition) von Funktionen und Datenstrukturen. Beispiel sind Listen.

Datenstrukturen (Rekords) in JS

```
x = {
  a1 : ε,
  a2 : ε,
  ..
}
x.ai = ε(x.ai)
```

Rekords versa Arrays

- Rekords (Structures) besitzen benannte Feldelemente oder Attribute, die Reihenfolge der Element ist nicht festgelegt! Die Indizierung erfolgt über den Feldnamen.
- Arrays bestehen aus unbenannten Elemente mit einer festgelegten Reihenfolge. Die Indizierung erfolgt über einen numerischen Index der die Position des Elements in einem Array bestimmt.

Rekord

```
x = {
  a:1,
  b:2,
  c:3
}
x.a=0
sum=x.a+x.b.x.c
```

Array

```
x = [
  1,
  2,
  3
]
x[0]=0
sum=x[0]+x[1]+x[2]
```

Abstrakte Datentypen

Man unterscheidet grob zwei Klassen von Daten und Datenstrukturen:

1. Datenstrukturen die durch die Programmiersprache (und nicht durch Bibliotheken) als Kernelemente zur Verfügun gestellt werden, z.B.
   - Arrays
   - Records
   - Listen (Haskell)
   - Enumeration
2. Datenstrukturen die durch den Programmierer implementiert werden
   - Listen
   - Bäume
   - Graphen