

Inhalt

1. **Komplexität**
2. **Generische Algorithmen**
 - a. Gierige Algorithmen
 - b. „Teile und Herrsche“-Algorithmen
 - c. Backtracking
3. **Konstruktionsprinzipien**
 - a. Schrittweise Verfeinerung
 - b. Einsatz von Algorithmenmustern
 - c. Problemreduzierung durch Rekursion
4. **Verifikation**



Algorithmen & Datenstrukturen I

WS 2002/03

Prof. Dr. Stefan Fischer

5. Algorithmenkonstruktion I

7.1 Komplexität

Jedes Problem lässt sich *durch verschiedene Algorithmen und Datenstrukturen lösen*.

Welche soll man nehmen ?

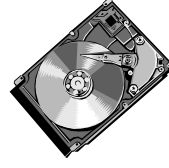
⇒ **Datenstrukturen verbrauchen Speicher**
 ⇒ **Algorithmen verbrauchen Rechenzeit**

– Verschiedene Algorithmen können für dasselbe Problem *sehr unterschiedlichen* Aufwand erfordern.

– Direkte Angabe von Zeitbedarf (ms) oder Platzbedarf (kByte) ist *systemabhängig* und deshalb zur Charakterisierung von Algorithmen ungeeignet.

– *Abstrakte* Komplexitätsmaße werden benötigt!

– Zeitbedarf steht im Mittelpunkt des Interesses.



Grundbegriffe

- **Umfang n eines Problems:**
Anzahl der Eingabewerte, Größe der Eingabewerte, ...
 - **Aufwand $T(n)$ eines Algorithmus zur Lösung des Problems mit dem Umfang n :**
Anzahl Zeiteinheiten (z.B. auch Operationen wie Addition, Multiplikation, Vergleich,...) bzw. Anzahl Speichereinheiten
 - **ungünstigster Aufwand (worst case)**
 - **mittlerer Aufwand (average case)**
 - **günstigster Aufwand (best case)**
- } asymptotisch für
 $n \rightarrow \infty$
- **Komplexität** eines Problems:
geringstmöglicher Aufwand mit irgendeinem Algorithmus, der das Problem löst
 - Anmerkung: ein Algorithmus terminiert, wenn $\forall n: T(n) < \infty$



Beispiel I (1/3)

Suchen einer Zahl x in einer Folge von n Zahlen

Lösung:

1. Zahlenfolge ist in Array a der Länge n gespeichert
2. Zahl wird linear gesucht

Algorithmus (Java):

$i = 0$; while ($i < n$ & $a[i] != x$) $i = i + 1$;

(ist nach Ausführung $i < n$, so ist die Zahl an Position i vorhanden)

Komplexität:

- ist x vorhanden (an Position i), so benötigt der Algorithmus $i+1$ Schritte (=elementare Anweisungen).
- ist x nicht vorhanden, so benötigt der Algorithmus $n+1$ Schritte



Technische Universität
Braunschweig

5-5

Beispiel I (2/3)

Damit ist der schlechteste Fall (Zahl ist nicht vorhanden) bzw. ungünstigste Aufwand in Bezug auf die Laufzeit erkannt:

Im schlechtesten Fall benötigt der Algorithmus $n+1$ Schritte

Der beste Fall (Zahl liegt an erster Stelle) bzw. günstigste Aufwand ist ebenfalls leicht zu beschreiben:

Im besten Fall benötigt der Algorithmus einen Schritt

Für eine Aussage über den mittleren Aufwand (Zahl wird an Position i gefunden) hat die obige Aussage noch zu viele Parameter: sie ist von der gesuchten Zahl bzw. ihrer Position im Array abhängig.

Lösung: Einbeziehung von Aussagen über die Häufigkeit, mit der eine Zahl an Position $1, 2, \dots, n$ gefunden wird:

Annahme: die Zahl wird mit gleicher *Wahrscheinlichkeit* an Position j ($0 \leq j \leq n-1$) gefunden (Gleichverteilung): sie beträgt für jedes j $1/n$



Technische Universität
Braunschweig

5-6

Beispiel I (3/3)

Unter der Gleichverteilungsannahme lässt sich der *mittlere Aufwand* durch asymptotische Betrachtung bestimmen:

- Gleichverteilung heißt, dass bei N Suchvorgängen die gesuchte Zahl in N/n Fällen an Position j gefunden wird.
- Damit werden für N Suchvorgänge insgesamt

$$M = \frac{N}{n} \cdot 1 + \frac{N}{n} \cdot 2 + \dots + \frac{N}{n} \cdot n \quad \text{Schritte benötigt.}$$

- Die Auswertung ergibt:

$$M = \frac{N}{n} \cdot (1 + 2 + \dots + n) = \frac{N}{n} \cdot \frac{n \cdot (n+1)}{2} = N \cdot \frac{n+1}{2}$$

- Für eine Suche werden *im Mittel* demnach $S = \frac{M}{N}$ Schritte benötigt, d.h. $S(n) = \frac{n+1}{2}$ beschreibt den mittleren Aufwand.



Technische Universität
Braunschweig

5-7

Das O-Kalkül (1/11)

Bestimmung der "Größenordnung" des Aufwands bzw. der Komplexität. Präzises Rechnen mit "ungefähr" ...

- es ist *nicht wesentlich*, ob der Aufwand $10 \cdot n$ oder $42 \cdot n$ beträgt (beide Funktionen sind linear).
- der Unterschied zwischen $42 \cdot n$, $42 \cdot n^2$ und $42 \cdot 2^n$ ist *allerdings relevant*.

Einführung von Komplexitätsklassen:

- die Algorithmen in einer Klasse haben hinsichtlich ihrer Komplexität eine gemeinsame obere oder untere Schranke -oder beides.
- das Zeitverhalten für kleine n ist irrelevant. Das asymptotische Verhalten für $n \rightarrow \infty$ ist von Interesse.



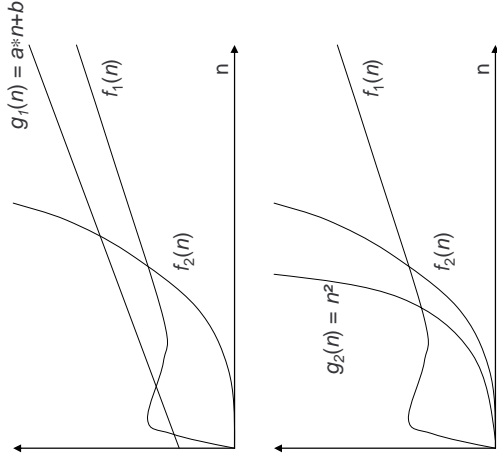
Technische Universität
Braunschweig

5-8

Das O-Kalkül (2/11)

Beispiele

- Klasse linearer Komplexitätsfunktionen**
Das Wachstumsverhalten von f_1 lässt sich durch g_1 gut nach oben abschätzen.
Das Wachstumsverhalten von f_2 hingegen offensichtlich nicht.
- Klasse quadratischer Komplexitätsfunktionen**
Das Wachstumsverhalten von f_1 lässt sich auch durch g_2 nach oben abschätzen, allerdings sehr ungenau.
Das Wachstumsverhalten von f_2 ist durch g_2 recht gut nach oben abgeschätzt.



Technische Universität Braunschweig

5-9

Das O-Kalkül (3/11)

Definition:

Die **Komplexitätsklassen** einer Funktion $g : \mathbb{N} \rightarrow \mathbb{N}$ sind

- $O(g(n)) = \{f(n) \mid \exists c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0: 0 \leq f(n) \leq c * g(n)\}$
obere Schranke: f wächst höchstens so schnell wie g
- $\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0: 0 \leq c * g(n) \leq f(n)\}$
untere Schranke: f wächst mindestens so schnell wie g
- $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, n_0 \in \mathbb{N}, \forall n \geq n_0: 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)\}$
untere und obere Schranke: f wächst i.w. so schnell wie g

Satz 7-1:

- $f \in \Theta(g) \Leftrightarrow f \in O(g) \wedge f \in \Omega(g)$
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$

Anmerkung: es gibt noch die Klassen o und ω (s. Goos I, S. 313)

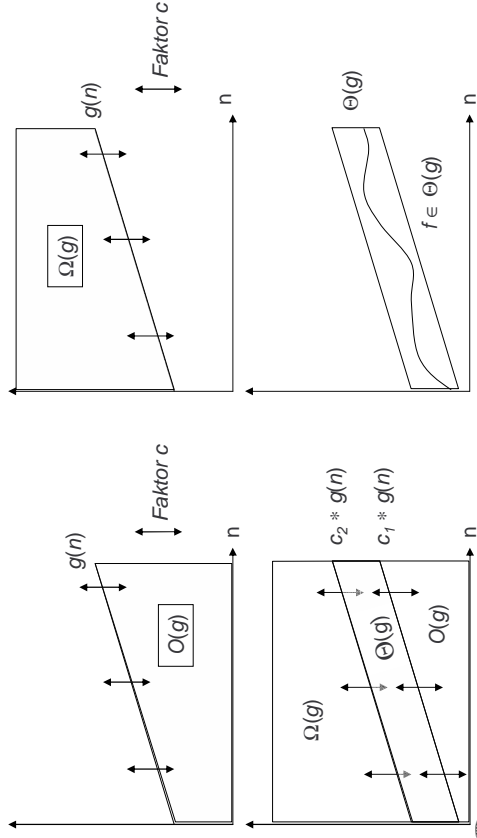


Technische Universität Braunschweig

5-10

Das O-Kalkül (4/11)

Veranschaulichung (vereinfacht für den linearen Fall)



Technische Universität Braunschweig

5-11

Das O-Kalkül (5/11)

Beispiel zur Klassifizierung eines Algorithmus:

Die Komplexitätsfunktion unseres Suchalgorithmus lautet $S(n) = \frac{n+1}{2}$

$\exists c := 1 > 0, n_0 := 2 \in \mathbb{N}, \forall n \geq n_0: 0 \leq S(n) \leq c * n = n =: g(n)$

$\Rightarrow S(n) \in O(g(n)) = O(n)$

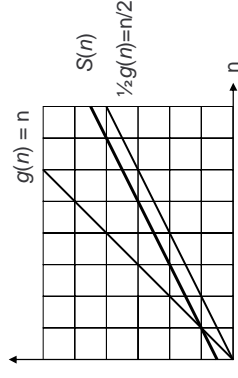
$\exists c := 1/2 > 0, n_0 := 1 \in \mathbb{N}, \forall n \geq n_0: 0 \leq c * n = n/2 = 1/2 * g(n) \leq S(n)$

$\Rightarrow S(n) \in \Omega(g(n)) = \Omega(n)$

Es gilt $f \in \Theta(g) \Leftrightarrow f \in O(g) \wedge f \in \Omega(g)$

$\Rightarrow S(n) \in \Theta(g(n)) = \Theta(n)$

Sprechweise: „der Algorithmus hat lineare Komplexität“



Technische Universität Braunschweig

5-12

Das O-Kalkül (6/11)

Gebräuchliche Komplexitätsklassen:

i	$O(kf_i)$	→ Erläuterung
1.	$O(1)$	→ höchstens konstanter Aufwand
2.	$O(\log n)$	→ höchstens logarithmischer Aufwand → <i>praktisch kaum mehr als konstanter Aufwand</i>
3.	$O(n)$	→ höchstens linearer Aufwand
4.	$O(n \log n)$	→ <i>praktisch kaum mehr als linearer Aufwand</i>
5.	$O(n^2)$	→ höchstens quadratischer Aufwand
6.	$O(n^k)$	→ höchstens polynomialer Aufwand
7.	$O(2^n)$	→ höchstens exponentieller Aufwand

Dabei gilt, dass $O(kf) \subseteq O(kf_{i+1})$ bzw. $kf_i \in O(kf_{i+1})$ für $1 \leq i \leq 6$

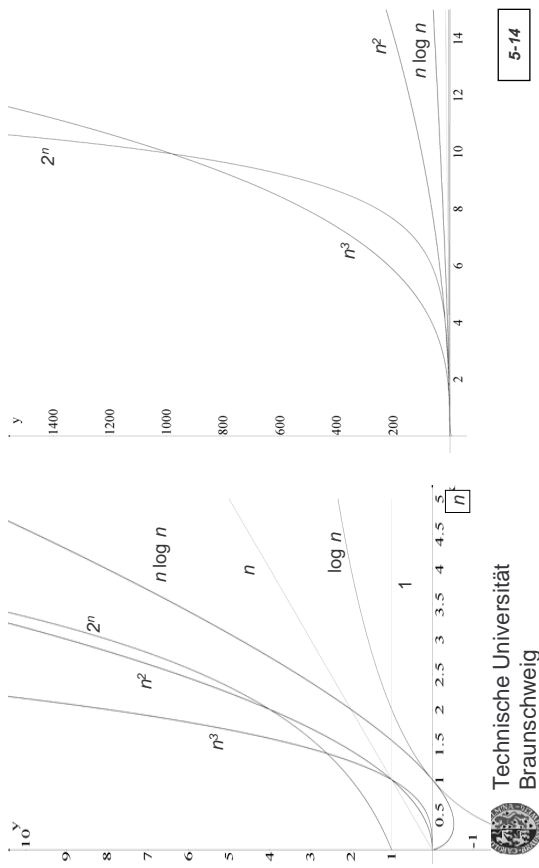


Technische Universität
Braunschweig

5-13

Das O-Kalkül (7/11)

Funktionen zu den gebräuchlichen Komplexitätsklassen:



5-14

Das O-Kalkül (8/11)

Exemplarische Werte (näherungsweise)

$n =$	1	10	100	1 000	10 000
$\log_2 n \approx$	0	3	7	10	13
$n \log_2 n \approx$	0	33	664	9 966	132 877
$n^2 =$	1	100	10 000	1 000 000	100 000 000
$2^n \approx$	2	10^3	10^{30}	10^{301}	10^{3010}

Ungerechnet auf Zeitbedarf der Berechnungen
(wobei der Zeitbedarf eines Schrittes 1 μ s betrage)

G	T=1 Min.	1 Std.	1 Tag	1 Woche	1 Jahr
n	$6 \cdot 10^7$	$3,6 \cdot 10^9$	$8,6 \cdot 10^{10}$	$6 \cdot 10^{11}$	$3 \cdot 10^{13}$
n^2	7750	$6 \cdot 10^4$	$2,9 \cdot 10^5$	$7,9 \cdot 10^5$	$5,6 \cdot 10^6$
n^3	391	1530	4420	8450	31600
2^n	25	31	36	39	44



Technische Universität
Braunschweig

5-15

Das O-Kalkül (9/11)

Notation und Rechenregeln (1/3):

Statt $g \in O(f)$ wird meist $g = O(f)$ geschrieben, ebenso für Ω und Θ .
Warum? Folgender Satz...

Satz 7-2: Sei $c \in \mathbb{N}$ und seien $f, g: \mathbb{N} \rightarrow \mathbb{N}$, sowie $h \in \Phi(f)$. Dann gilt für $\Phi \in \{O, \Omega, \Theta\}$

$$h + g \in \Phi(f + g)$$

$$h * g \in \Phi(f * g)$$

$$h / g \in \Phi(f / g)$$

$$h \pm c \in \Phi(f)$$

$$c * h \in \Phi(f)$$

Für $h \in \Theta(f)$ gilt auch

$$g / h \in \Theta(g / f)$$



Technische Universität
Braunschweig

5-16

Das O-Kalkül (10/11)

Notation und Rechenregeln (2/3):

Aufgrund von Satz 7-2 definieren wir für $\Phi \in \{O, \Omega, \Theta\}$:

$$\Phi(f) \pm g := \Phi(f \pm g)$$

$$\Phi(f) + \Phi(g) := \Phi(f + g)$$

$$c * \Phi(f) := \Phi(f)$$

$$\Phi(f) * \Phi(g) := \Phi(f * g)$$

$$\Phi(f) / \Phi(g) := \Phi(f / g)$$

Schließlich gilt:

Satz 7-3: Die Klassen O, Ω, Θ sind transitiv: sei $\Phi \in \{O, \Omega, \Theta\}$

$$f \in \Phi(g), g \in \Phi(h) \Rightarrow f \in \Phi(h)$$

$$O, \Omega, \Theta \text{ sind reflexiv } (f \in \Phi(f))$$

$$\Theta \text{ ist symmetrisch } (f \in \Theta(g) \Leftrightarrow g \in \Theta(f))$$



Technische Universität
Braunschweig

5-17

Das O-Kalkül (11/11)

Notation und Rechenregeln (3/3):

Satz 7-2 (und das Wissen über die „Ordnung“ der Komplexitätsklassen; s.o.) vereinfacht das „Rechnen“ mit dem O-Kalkül.

Beispiel:

$$3n^3 - 4n + 5 \log n - 6 = 3n^3 - 4n + 5 \log n + O(1)$$

$$= 3n^3 - 4n + O(\log n)$$

$$= 3n^3 + O(n)$$

$$= O(n^3)$$

Solche „Gleichungen“ dürfen nur von links nach rechts gelesen werden: die Abschätzung wird immer größer.



Technische Universität
Braunschweig

5-18

Beispiel II (1/3)

Sortieren von n Zahlen (o.ä.) durch Einfügen: Insertion Sort

Algorithmus:

Elemente werden nacheinander der gegebenen unsortierten Liste entnommen und in eine anfangs leere neue Liste an der richtigen Stelle eingefügt.

c_0 sei die Zeit für das Sortieren der leeren Liste, und c_1 die für das Herausnehmen des nächsten Elements aus der unsortierten Liste.

Komplexität: Die Komplexitätsfunktion für den average case lässt sich rekursiv bestimmen

$$- T_{\text{isort}}(0) = c_0$$

$$- T_{\text{isort}}(n) = T_{\text{isort}}(n-1) + c_1 + T_{\text{isort}}(n-1)$$

wobei $T_{\text{isort}}(n)$ den mittleren Aufwand beschreibt, ein Element entsprechend seiner Ordnung in eine Liste einzufügen.



Technische Universität
Braunschweig

5-19

Beispiel II (2/3)

Analog zu Beispiel I (Suchen eines Elements in einer Liste) ist die

Komplexität des Einfügens (average case) mit $T_{\text{insert}}(n) = c \cdot \frac{n+1}{2}$

anzugeben (mit $c = \text{Zeiteinheit für Vergleich und Einfügeoperation}$). Hierbei liegt die Annahme zugrunde, dass jede Einfügeposition gleich

wahrscheinlich ist.

Damit:

$$- T_{\text{isort}}(1) = c_0 + c_1 + T_{\text{isort}}(0) = c_0 + c_1 + c/2$$

$$- T_{\text{isort}}(2) = T_{\text{isort}}(2-1) + c_1 + T_{\text{isort}}(2-1)$$

$$= c_0 + c_1 + c/2 + c_1 + T_{\text{isort}}(1) = c_0 + 2 * c_1 + c/2 * (1+2)$$

$$- T_{\text{isort}}(3) = T_{\text{isort}}(3-1) + c_1 + T_{\text{isort}}(3-1)$$

$$= c_0 + 2 * c_1 + c/2 * (1+2) + c_1 + T_{\text{isort}}(2)$$

$$= c_0 + 3 * c_1 + c/2 * (1+2+3)$$



Technische Universität
Braunschweig

5-20

Beispiel II (3/3)

- $T_{\text{isort}}(1) = c_0 + c_1 + c/2$
- $T_{\text{isort}}(2) = c_0 + 2 * c_1 + c/2 * (1+2)$
- $T_{\text{isort}}(3) = c_0 + 3 * c_1 + c/2 * (1+2+3)$

Durch Induktion über $n \geq 0$:

$$= O(1) + O(n) + O(n^2)$$

$$= O(n^2)$$

D.h. Sortieren durch Einfügen hat im Mittel quadratischen Aufwand.



P und NP (1/2)

Betrachtet man die Entwicklung der den entsprechenden Komplexitätsklassen zugrundeliegenden Funktionen, so sind zwei Klassen von Komplexitäten zu unterscheiden:

1. Algorithmen mit polynomiellem Aufwand $O(n^k)$
=> diese sind auch bei großen Eingaben noch effizient zu berechnen
2. Algorithmen mit exponentiellem Aufwand $O(k^n)$
=> die Berechnung dieser Probleme entzieht sich bei größer werdender Eingabe rasch der praktischen Berechenbarkeit

Definition P und NP:

1. Alle Probleme, die mit Hilfe deterministischer Algorithmen in polynomieller Zeit gelöst werden können gehören zur Klasse **P**
2. Alle Probleme, die nur mit Hilfe nichtdeterministischer Algorithmen in polynomieller Zeit gelöst werden können gehören zur Klasse **NP** (für nichtdeterministisch polynomiell)



P und NP (2/2)

Offenbar gilt: $P \subseteq NP$

Offene Frage: gilt $P = NP$?

Beispiel für eine Problem aus NP: Erfüllbarkeitsproblem (SAT)

Gegeben: Aussagenlogische Formel a mit n Variablen und logischen Operatoren \neg, \wedge, \vee sowie Klammern

Gesucht: Aussage, ob Belegung existiert, die a erfüllt.

Für SAT gibt es derzeit keine polynomielle Lösung: $SAT \in NP$.

Anmerkung: Es gibt eine Klasse von Problemen aus NP (die sogenannten NP-vollständigen Probleme), für die gilt: kann eines dieser Probleme in polynomieller Zeit gelöst werden, dann können auch alle anderen in polynomieller Zeit gelöst werden und es gilt $P=NP$.

SAT gehört zur Klasse der NP-vollständigen Probleme ebenso wie das Problem des Handlungsreisenden (TSP; s.u.)



Analyse von Algorithmen (1/2)

Wie die Laufzeitkomplexität eines Algorithmus bestimmen?

Folgende Regeln zur Schätzung bei konkreten Programmen:

1. for-Schleifen (while/until-Schleifen analog + Laufzeit der Bedingungsprüfung)

- Laufzeit := max. Laufzeit der inneren Anweisung * Anzahl der Iterationen
- Beispiel: for ($i=1; i \leq n; i++$) $a[i]=0$;

Da die Anzahl der Iterationen n beträgt und die innere Anweisung den Aufwand $O(1)$ hat, ergibt sich die Laufzeitschätzung zu $n * O(1) = O(n)$

2. Geschachtelte for-Schleifen

- Laufzeit := max. Laufzeit der innersten Anweisung * Produkt der Iterationen aller Schleifen
- Beispiel: for ($i=1; i \leq n; i++$)
for ($j=1; j \leq n; j++$) $k=k+b[i,j]$;

Der Aufwand der inneren Schleife ist wiederum $O(1)$. So ergibt sich der Gesamtaufwand zu $n * n * O(1) = O(n^2)$



Analyse von Algorithmen (2/2)

3. Nacheinanderausführung

- Laufzeit = Summe der Laufzeiten unter Weglassung konstanter Faktoren und Reduzierung auf den höchsten Exponenten.
- Beispiel: for ($j=1; k=n; j++$) a[j]=0; // Laufzeit $O(n)$; s.o.
for ($i=1; k=n; i++$) // Laufzeit $O(n^2)$; s.o.
for ($j=1; k=n; j++$) $k=k+b[i,j]$;

Die Laufzeitabschätzung ergibt sich zu $O(n^2) + O(n) = O(n^2)$.

4. if-else-Bedingungen

- Laufzeit = Laufzeit der Auswertung der Bedingung + $\max\{\text{Laufzeit Alternative 1, Laufzeit Alternative 2}\}$
- Beispiel: if ($k < 50$) $k = k + 10$;
else for ($j=1; j \leq n; j++$) $k=k+a[j]$;

Der Aufwand der Auswertung und der ersten Alternative ist $O(1)$. Die zweite Alternative hat eine Laufzeit in $O(n)$. So ergibt sich der Gesamtaufwand zu $O(1) + \max\{O(1), O(n)\} = O(n)$



7.2 Generische Algorithmen

Idee: Entwicklung algorithmischer Muster für eine Klasse von Problemen zwecks Wiederverwendung.

Anwendung:

1. Wiedererkennen der generischen Problemstruktur in einem konkreten Problem.
2. Anwenden des Algorithmenmusters auf konkretes Problem durch Anpassung.

Behandelt werden:

- a. Gierige Algorithmen
- b. „Teile und Herrsche“-Algorithmen
- c. Backtracking



Gierige (Greedy-) Algorithmen (1/19)

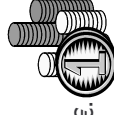
Ziel: Lösen einer Optimierungsaufgabe

- **Problemstellung:**
 1. es gibt eine *endliche Menge* von Eingabewerten
 2. es gibt eine *Menge von (Teil-)Lösungen*, die aus Eingabewerten aufgebaut sind
 3. es gibt eine Bewertungsfunktion für (Teil-)Lösungen
 4. die Lösungen lassen sich schrittweise aus Teillösungen, beginnend bei der leeren Lösung, durch Hinzunahme von Eingabewerten aufbauen
 5. gesucht wird eine (die) optimale Lösung
- **Vorgehensweise:**

nimm (gierig) immer das best bewertete Stück
→ lokales Optimum wird gewählt.



Gierige (Greedy-) Algorithmen (2/19)



Beispiel 1: Geldwechseln

- als Münzen gibt es 1, 2, 5, 10, 20 und 50 Cent sowie 1 und 2 €.
- Wechselgeld soll mit möglichst wenig Münzen ausgezahlt werden.
z.B. 1,42 € → $1 \times 1 \text{ €} + 2 \times 20 \text{ Cent} + 1 \times 2 \text{ Cent}$
- **Allgemein:** wähle als nächstes eine *größtmögliche* Münze, um schnell voran zu kommen (Bewertungsfunktion = Wert der Münze)
- In unserem Münzsystem gibt dies immer ein globales Optimum. Dies muss bei "unkonventionellen" Münzsystemen nicht so sein! Hätten wir z.B. 1, 5 und 11 Cent-Münzen zur Verfügung und sollten 15 Cent herausgeben, so hätten wir folgende Situation:

- gierig : $11+1+1+1+1 \rightarrow 5$ Münzen
- optimal : $5+5+5 \rightarrow 3$ Münzen



Gierige (Greedy-) Algorithmen (3/19)

Algorithmenschema (Java-Pseudocode)

```
public Ergebnismenge löseGierig(Eingabemenge E) {
// E steht für die Menge von Eingabewerten, L für die Lösungsmenge.
    Ergebnismenge L = new Ergebnismenge();
    Eingabeelement x; // gleichzeitig Ergebniselement
    while (! E.isEmpty()) {
        if (komplett(L)) return L; // komplett() ist Boole'sche Funktion
            // zur Bestimmung, ob eine vollständige Lösung gefunden ist
        x = nächsterKandidat(E); E.delete(x); // nächsterKandidat() wird
            // nicht weiter ausgeführt und bestimmt i.d.R. das Maximum oder
            // Minimum bzgl. einer gegebenen Ordnungsrelation
        if (geeignet(x,L)) L.insert(x); // geeignet() liefert true genau dann,
            // wenn {x} ∪ L Teilmenge einer Lösung sein kann
    }
}
```



Gierige (Greedy-) Algorithmen (4/19)

Beispiel 2 (1/2): Bedienreihenfolge im Supermarkt mit Greedy ermitteln

Problem: n Kunden warten vor einer Kasse.

- Der Bezahlvorgang von Kunde i dauert c_i Sekunden.
- Welche Reihenfolge der Bedienung (bzw. welche Bewertungsfunktion) der Kunden führt zur Minimierung der mittleren Verweilzeit?

Gesamtbedienzeit: ist konstant

Analyse: die mittlere Verweilzeit ist



Gierige (Greedy-) Algorithmen (5/19)

Beispiel 2 (2/2): Bedienreihenfolge im Supermarkt mit Greedy ermitteln

Mittlere Verweilzeit pro Kunde:

- steigt, wenn Kunden mit langer Bedienzeit vorgezogen werden
- sinkt, wenn Kunden mit kurzer Bedienzeit zuerst bedient werden
- wird minimal, wenn die Kunden nach c_i aufsteigend sortiert werden.

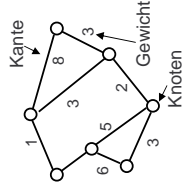
Konsequenzen:

- Greedy-Algorithmus geeignet.
- Bewertungsfunktion bildet Kunden auf Bedienzeit ab.
- Die Funktion zur Bestimmung des nächsten Kandidaten wählt den Kunden mit minimaler Bedienzeit.
- (Supermarkt wird in Bälle wg. Kundenaufstand geschlossen)

Frage: Strategie für Prozessorzeitvergabe geeignet?



Gierige (Greedy-) Algorithmen (6/19)



Beispiel 3: Minimal spannende Bäume (1/4)

Gegeben: ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, d)$.

- V ist eine endliche Knotenmenge, (engl. Vertices)
- $E \subseteq V^2$ ist eine Kantenmenge, (engl. Edges)
- $d: E \rightarrow \mathbb{R}^+$ gibt jeder Kante v ein "Gewicht" $d(v) > 0$.
- ungerichtet heißt $(x,y) \in E \Rightarrow (y,x) \in E$.
- zusammenhängend heißt

$\forall x_i, y_j \in V \exists x_1, \dots, x_k \in V (x, x_1), (x_1, x_2), \dots, (x_k, y) \in E$

Wozu Graphen? Z.B.

- zur Modellierung von Wegnetzen (Knoten=Städte, Kante=Verbindung, Gewicht=Kosten einer Reise) > TSP
- zur Modellierung von Kommunikationsnetzen (Knoten=Kommunikationsendpunkte, Kante=Verbindung, Gewicht=Kosten der Verbindung)



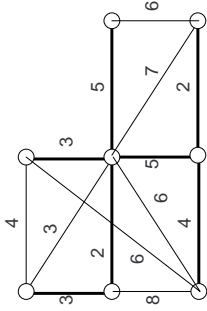
Gierige (Greedy-) Algorithmen (7/19)

Beispiel 3: Minimal spannende Bäume (2/4)

Gesucht: ein minimal spannender Baum zu G , d.h. eine minimale Teilmenge $E_{\min} \subseteq E$ der Kanten, so dass $G_{\min} = (V, E_{\min}, d)$ zusammenhängend und die Summe der Kantengewichte *minimal* ist.

In einer Lösung sind keine Zyklen enthalten, da sonst noch eine Kante herausgenommen werden könnte. *Die Lösung ist also ein Baum.*

Beispiel:



Die Kosten des minimalen spannenden Baumes sind:

$$2+2+3+3+4+5+5 = 24$$

Die Lösung ist nicht eindeutig!

Sehen Sie eine andere?



Gierige (Greedy-) Algorithmen (8/19)

Beispiel 3: Minimal spannende Bäume (3/4)

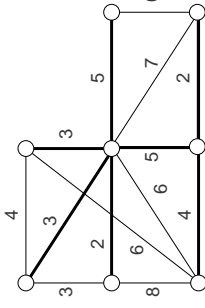
Kruskals gieriger Algorithmus zur Ermittlung eines minimal spannenden Baumes (1956): selektiere fortwährend eine verbleibende Kante mit geringstem Gewicht, die keinen Zyklus erzeugt, bis alle Knoten verbunden sind

Beispiel:

Die Kosten (dieses) minimalen spannenden Baumes sind: $2+2+3+3+4+5+5 = 24$

Anmerkungen:

- Nach Wahl der Kanten 2,2,3 und 3 dürfte die verbleibende 3 nicht gewählt werden, da ansonsten ein Zyklus entstehen würde.
- Eine eindeutige Lösung ist immer dann vorhanden, wenn alle Gewichte verschieden sind.



Gierige (Greedy-) Algorithmen (9/19)

Beispiel 3: Minimal spannende Bäume (4/4)

Satz 7-4: Kruskals Algorithmus erzeugt immer einen minimalen spannenden Baum.

Beweis: Sei G der gegebene Graph, und sei B ein minimaler spannender Baum von G . Der Satz folgt aus der folgenden Aussage:

Ist C ein Teilgraph von B , dann gibt es eine "billigste" von C ausgehende Kante b , die in B liegt.

Annahme: Keine der billigsten von C ausgehenden Kanten liegt in B .

Sei b eine der billigsten von C ausgehenden Kanten.

Wird B um b ergänzt, entsteht ein Zyklus, auf dem eine weitere Kante b' von C mit $b \neq b'$ liegen muss. b ist billiger als b' .

Wird nun in B b' gegen b ausgetauscht entsteht ein spannender Baum B' , der billiger ist.

Widerspruch zur Eigenschaft von B minimaler spannender Baum zu sein!



Gierige (Greedy-) Algorithmen (10/19)

Implementierung Alg. von Kruskal (1/2): Modellskizze

```

class Graph
    addNode(Vertex): Boolean
    connect(Vertex, Vertex): Boolean
    kruskal(): EVK_Set
    edges(): EVK_Set
  
```

```

class Vertex
    0..n vertices
  
```

```

class Edge
    weight(): float
  
```

edges 0..n

nodes 2

vertices 0..n

weight(): float



Gierige (Greedy-) Algorithmen (11/19)

Implementierung (2/3): Java-Pseudocode

```
public EVK_Set kruskal() {
    EVK_Set E = edges(); // Kopie der Kantenmenge
    EVK_Set L = new EVK_Set();
    Edge x;
    while (! E.isEmpty()) {
        if (alleKnotenEnthalten(L)) return L;
        x = findeKanteMitGeringstemGewicht(E); E.delete(x);
        if (ergänzungOhneZyklusMöglich(x,L)) L.insert(x);
    }
}
```

Zu ergänzen sind die Hilfsfunktionen alleKnotenEnthalten(EVK_Set): boolean, findeKanteMitGeringstemGewicht(EVK_Set): GraphEdge und ergänzungOhneZyklusMöglich(GraphEdge, EVK_Set): boolean



Gierige (Greedy-) Algorithmen (12/19)

Implementierung (3/3): Laufzeit

Die Laufzeit des oben angegebenen Algorithmus ist nicht besonders effizient!

Eine Verbesserung: Vor Start des Algorithmus Sortierung der Kantenmenge nach Gewicht in einer Liste und sukzessives Durchwandern der Liste (> *findeKanteMitGeringstemGewicht()*) wird dadurch sehr einfach)

Sortierung gelingt bei effizienter Implementierung (>AuD II) in $O(k \log k)$ wodurch sich die Gesamtlaufzeit von Kruskal's Algorithmus ebenfalls zu $O(k \log k)$ ergibt (k = Anzahl der Kanten; sind alle Knoten mit allen anderen verbunden ist $k \equiv n^2$).

Asymptotisch bessere Verfahren (bei vollständigen Graphen) mit einem Aufwand von $O(n^2)$ sind möglich: Algorithmus von Prim (1957)



Gierige (Greedy-) Algorithmen (13/19)

Algorithmus von Prim (1/4)

Ansatzpunkt: Verbesserung der Größe der pro Durchlauf betrachteten Kantenmenge E bzw. von *findeKanteMitGeringstemGewicht(E)*

Idee: Beschränkung der Suche auf Teilmenge V von E , wobei

1. V enthält immer die billigste aus R

(=bisher berechneter Teilgraph) ausgehende Kante

2. V sollte möglichst klein sein

3. V sollte einfach zu ändern sein (Forderung 1 ist zu garantieren !)

Annahmen:

1. Gewichte sind *verschieden* (die Lösung ist dann eindeutig),
2. der Graph ist *vollständig*: je zwei Knoten sind durch eine Kante verbunden (notfalls können Kanten von so hohem Gewicht hinzugefügt werden, dass sie sicher nicht in der Lösung auftauchen)

Technische Universität
Braunschweig

Gierige (Greedy-) Algorithmen (14/19)

Algorithmus von Prim (2/4)

Zwei Möglichkeiten:

1. V enthält für jeden Knoten r in R die billigste von r aus R herausführende Kante

2. V enthält für jeden Knoten a außerhalb R die billigste von a in R hineinführende Kante



$\vdots = V$ (Alternative 1)



$\vdots = V$ (Alternative 2)

Alternative 1 ist änderungsaufwendig, da mehrere Kanten zu einem externen Knoten führen können (ggf. müssen alle Kanten in V ersetzt werden).
=> Variante 2



Gierige (Greedy-) Algorithmen (15/19)

Algorithmus von Prim (3/4)

Erste Formalisierung:

```
public EVK_Set prim() {
    Vertex r = nodes.first(); // Auswahl eines beliebigen Knotens
    // hier: der erste (indizierbare Datenstruktur vorausgesetzt)
    EVK_Set V = <alle n-1 nach r führenden Kanten>
    EVK_Set L = new EVK_Set();
    for (int i=1; i<=n-1; i++) {
        x = findeKanteMitGeringstemGewicht(V); V.delete(x);
        L.insert(x); // Endknoten von x sei a
        <Ändere V so, dass Eigenschaft 1 (s.o.) erhalten bleibt>
    }
    return L;
}
```



Gierige (Greedy-) Algorithmen (16/19)

Algorithmus von Prim (4/4)

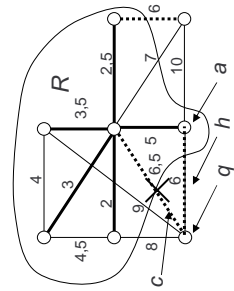
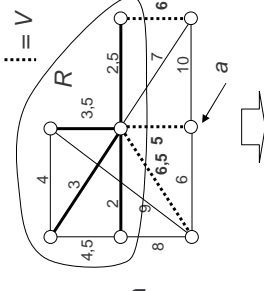
Zu spezifizieren bleibt: <Ändere V so, dass Eigenschaft 1 (s.o.) erhalten bleibt>

Es gilt: Jeder noch nicht in R enthaltene Knoten q hat die billigste Verbindung nach R wie zuvor oder aber mit a

Lösung für „Ändere V...“:

...
for (<alle mit a verbundenen Knoten q außerhalb R>)
for (<alle Kanten c in V mit Endpunkt q>)
if (<(Kante h von q nach a), weight() < c.weight()>)
<in Versetze c durch h>

Anmerkung: Änderung von V lässt sich in $O(n)$ Schritten durchführen



Gierige (Greedy-) Algorithmen (17/19)

Matroide (1/3)

Greedy-Algorithmen liefern *nicht immer* eine optimale Lösung (s.o.).
Frage: wann liefert ein Greedy-Algorithmus eine optimale Lösung ?
Eine Antwort liefert die Theorie der *bewerteten Matroide*.
Definition: Ein *Matroid* ist ein Paar $M = (U, I)$ mit folgenden Eigenschaften:
 1. U ist eine endliche nichtleere Menge
 2. I ist eine nichtleere Familie von Teilmengen von U (*unabhängige Mengen*)
 3. sind $A \subseteq B$ und $B \in I$, so ist $A \in I$ (*Vererbungseigenschaft*)
 4. sind $A, B \in I$ und $|A| < |B|$, so gibt es ein $x \in B \setminus A$: $A \cup \{x\} \in I$

Eine unabhängige Menge $A \in I$ heißt *maximal*, wenn sie keine Erweiterung in I besitzt.



Gierige (Greedy-) Algorithmen (18/19)

Matroide (2/3)

Definition:

- Ein Matroid M heißt *bewertet*, wenn es eine Bewertungsfunktion $w: U \rightarrow \mathbb{R}^+$ gibt.

- $w(A) =$ ist die Bewertung der Menge $A \subseteq U$.

- Eine Menge $A \subseteq U$ heißt *optimal*, wenn sie maximal ist und unter den maximalen Mengen minimalen Wert hat.

Satz 7-5: Ist ein bewertetes Matroid $M = (U, I, w)$ gegeben, so findet der folgende gierige Algorithmus eine optimale Menge $A \in I$, sortiere U nach steigenden Gewichten (Ergebnis ist eine Liste).

$\text{gierig}(U) = \text{gierig}'(\text{sort}(U), \emptyset)$

$\text{gierig}'(I, A) = A$ // *Alternativ A ist optimal: gierig'(h, A) = A*

$A \cup \{x\} \in I$: $\text{gierig}'(x : h, A) = \text{gierig}'(h, A \cup \{x\})$

$\text{gierig}'(x : h, A) = \text{gierig}'(h, A)$



Gierige (Greedy-) Algorithmen (19/19)

Matroide (3/3)

Beweis: Goos I (333-335)

Die **Komplexität** von Greedy-Algorithmen ist

$$O(n \log n + n * f(n)),$$

wenn

- $O(n \log n)$ der Aufwand für das Sortieren und
- $f(n)$ der Aufwand für den Test $A \cup \{x\} \in I$ ist.

Beispiel: Kruskal

- I ist die Menge aller Teilmengen der Kanten minimal spannender Bäume
- Eine optimale Lösung $A \in I$ entspricht einer maximalen Menge von Kanten (die einen Baum aufspannen) mit einem minimalen Gewicht.
- Achtung: Azyklizität der Elemente in I ist zusätzlich zu verlangen!



Technische Universität
Braunschweig

5-45

„Teile und Herrsche“-Algorithmen (1/13)

Prinzip der Rekursion:

Anwenden des gleichen Lösungsschemas auf ein (echt) reduziertes Teilproblem solange, bis ein Teilproblem direkt lösbar ist (Ende der Rekursion).

Prinzip von „Teile und Herrsche“ (engl. divide and conquer; lat. divide et impera):

1. Aufteilen des Gesamtproblems in mehrere kleinere Teilprobleme
2. Rekursives Bearbeiten der Teilprobleme

Voraussetzungen

1. Problem a muss in gleichartige Teilprobleme a_1, \dots, a_p zerlegbar sein
2. Teilprobleme müssen jeweils einen wesentlichen Beitrag zur Gesamtlösung liefern

Typische Beispiele: Sortierverfahren Mergesort, Quicksort (AuD II), ...



Technische Universität
Braunschweig

5-46

Gierige (Greedy-) Algorithmen (19/19)

Matroide (3/3)

Beweis: Goos I (333-335)

Die **Komplexität** von Greedy-Algorithmen ist

$$O(n \log n + n * f(n)),$$

wenn

- $O(n \log n)$ der Aufwand für das Sortieren und
- $f(n)$ der Aufwand für den Test $A \cup \{x\} \in I$ ist.

Beispiel: Kruskal

- I ist die Menge aller Teilmengen der Kanten minimal spannender Bäume
- Eine optimale Lösung $A \in I$ entspricht einer maximalen Menge von Kanten (die einen Baum aufspannen) mit einem minimalen Gewicht.
- Achtung: Azyklizität der Elemente in I ist zusätzlich zu verlangen!



Technische Universität
Braunschweig

5-45

„Teile und Herrsche“-Algorithmen (1/13)

Prinzip der Rekursion:

Anwenden des gleichen Lösungsschemas auf ein (echt) reduziertes Teilproblem solange, bis ein Teilproblem direkt lösbar ist (Ende der Rekursion).

Prinzip von „Teile und Herrsche“ (engl. divide and conquer; lat. divide et impera):

1. Aufteilen des Gesamtproblems in mehrere kleinere Teilprobleme
2. Rekursives Bearbeiten der Teilprobleme

Voraussetzungen

1. Problem a muss in gleichartige Teilprobleme a_1, \dots, a_p zerlegbar sein
2. Teilprobleme müssen jeweils einen wesentlichen Beitrag zur Gesamtlösung liefern

Typische Beispiele: Sortierverfahren Mergesort, Quicksort (AuD II), ...



Technische Universität
Braunschweig

5-46

Gierige (Greedy-) Algorithmen (19/19)

Matroide (3/3)

Beweis: Goos I (333-335)

Die **Komplexität** von Greedy-Algorithmen ist

$$O(n \log n + n * f(n)),$$

wenn

- $O(n \log n)$ der Aufwand für das Sortieren und
- $f(n)$ der Aufwand für den Test $A \cup \{x\} \in I$ ist.

Beispiel: Kruskal

- I ist die Menge aller Teilmengen der Kanten minimal spannender Bäume
- Eine optimale Lösung $A \in I$ entspricht einer maximalen Menge von Kanten (die einen Baum aufspannen) mit einem minimalen Gewicht.
- Achtung: Azyklizität der Elemente in I ist zusätzlich zu verlangen!



Technische Universität
Braunschweig

5-45

„Teile und Herrsche“-Algorithmen (3/13)

Vorgehensweise:

- löse hinreichend kleine Probleme a direkt: $Lösung(a)$
- zerlege großes Problem in Teilprobleme:
zerlege : $a \rightarrow (a_1, \dots, a_p)$
- löse jedes der Teilprobleme getrennt
– und zwar rekursiv nach dem gleichen Schema !
- setze die Teillösungen zur Gesamtlösung zusammen:
zusammen : $(Lösung(a_1), \dots, Lösung(a_p)) \rightarrow Lösung(a)$

Algorithmenschema (Imperativ, Java-Pseudocode):

```
public Lösung divideAndConquer(Problem P) {
    Lösung result = new Lösung();
    if ( Klein(P) ) return expliziteLösung( P );
    else {
        Lösung[] L = new Lösung[p]; // Array der p Teillösungen
        Problem[] TP = teileAuf( P ); // Array der p Teilprobleme
        L[1] = divideAndConquer( TP [1] );
        ...
        L[p] = divideAndConquer( TP [p] );
        result = setzeZusammen( L[1], ..., L[p] );
        return result;
    }
}
```



Technische Universität
Braunschweig

5-47



Technische Universität
Braunschweig

5-48

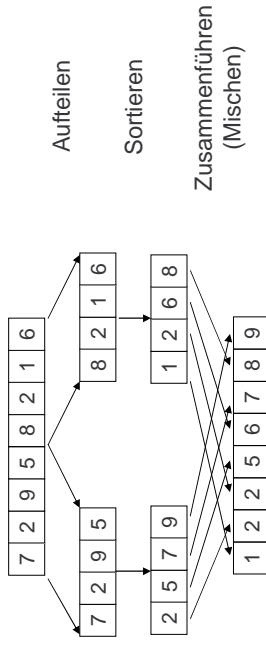
„Teile und Herrsche“-Algorithmen (4/13)

Beispiel 1: Mergesort (Sortieren durch Mischen) (1/3)

Problem: eine Folge von Zahlen ist zu sortieren

„Teile-und-Herrsche“-Lösung Mergesort:

1. Teile die Folge in zwei Teilfolgen auf
2. Sortiere (rekursiv) die zwei Teilfolgen
3. "Mische" die sortierten Folgen zu einer sortierten Folge



Technische Universität
Braunschweig

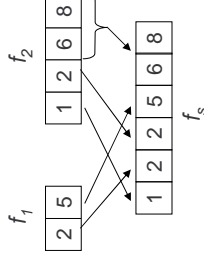
5-49

„Teile und Herrsche“-Algorithmen (5/13)

Beispiel 1: Mergesort (Sortieren durch Mischen) (2/3)

Algorithmaschema für das Mischen (Java-Pseudocode):

```
public Folge merge(Folge f1, Folge f2) {
    // Eingabe: zwei sortierte Folgen f1 und f2
    // Ausgabe: eine sortierte Folge fs
    Folge fs = new Folge(); // Erzeuge leere Folge
    while (!f1.isEmpty() & !f2.isEmpty())
        if (f1.getFirst() <= f2.getFirst())
            fs.addLast(f1.getFirstAndRemove());
        else
            fs.addLast(f2.getFirstAndRemove());
    if (!f1.isEmpty()) fs.addLastAllOf(f1);
    if (!f2.isEmpty()) fs.addLastAllOf(f2);
    return fs;
}
```



Technische Universität
Braunschweig

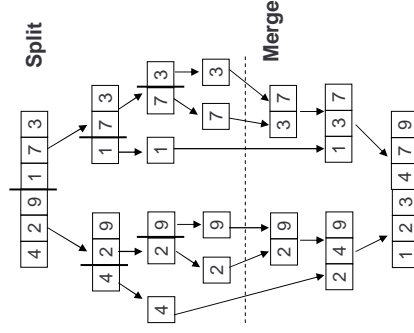
5-50

„Teile und Herrsche“-Algorithmen (6/13)

Beispiel 1: Mergesort (Sortieren durch Mischen) (3/3)

Algorithmaschema für Mergesort (Java-Pseudocode):

```
public Folge mergesort(Folge fE) {
    // Eingabe: eine unsortierte Folge fE
    // Ausgabe: eine sortierte Folge fs
    if (fE.length() == 1) return fE; // Abbruch,
    // wenn einelementige Folge erreicht
    else { // Teile fE in zwei Folgen f1 und f2;
        // Sortiere rekursiv Teilfolge f1
        f1 = mergesort(f1);
        // Sortiere rekursiv Teilfolge f2
        f2 = mergesort(f2);
        // Mische die sortierten Folgen f1 und f2
        // und gib das Ergebnis zurück
        return merge(f1, f2);
    }
}
```



Technische Universität
Braunschweig

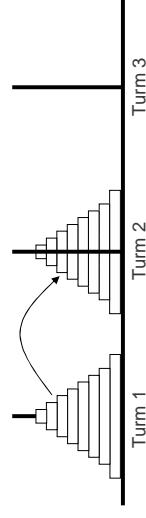
5-51

„Teile und Herrsche“-Algorithmen (7/13)

Beispiel 2 : Türme von Hanoi (1/5)

Der Legende nach versucht sich ein buddhistischer Mönchsorden seit Jahrhunderten (wenn nicht Jahrtausenden) an folgendem **Problem**:

- 64 goldene Scheiben verschiedener Größe sind aufeinandergestapelt, der Größe nach sortiert: immer nur kleinere Scheiben auf größeren.
- der gesamte Stapel soll Scheibe für Scheibe umgestapelt werden, so dass der neue Stapel am Ende genauso aussieht wie der alte.
- ein dritter Stapel darf zur Zwischenspeicherung benutzt werden, ansonsten dürfen die Scheiben nirgendwo anders abgelegt werden.
- auch in jedem Zwischenzustand dürfen nur kleinere Scheiben auf größeren liegen, niemals umgekehrt.



Turm 1: Ausgangsstapel
Turm 2: Zielstapel
Turm 3: Zwischenspeicher



Technische Universität
Braunschweig

5-52

„Teile und Herrsche“-Algorithmen (8/13)

Beispiel 2 : Türme von Hanoi (2/5)

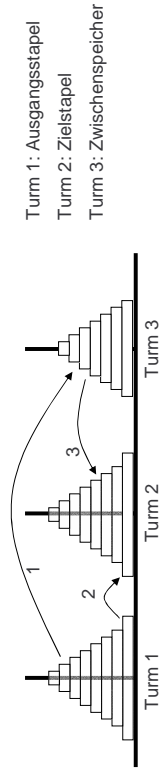
Gesucht ist ein Algorithmus, der dieses Problem löst – wenn es denn lösbar ist. Dazu muss der Algorithmus angeben, in welcher Reihenfolge welche Scheibe von wo nach wo zu bewegen ist.

... und wir suchen nach einem *generischen* Algorithmus für eine beliebige Anzahl n von Scheiben.

Lösungsidee mit schrittweiser Verfeinerung:

$n = 1$: bewege die Scheibe von Turm 1 zu Turm 2.

sonst: es gibt nur eine Möglichkeit, die unterste Scheibe von Turm 1 zu Turm 2 zu bekommen: die $n-1$ oberen Scheiben müssen zu Turm 3 bewegt werden.



Technische Universität
Braunschweig

5-53

„Teile und Herrsche“-Algorithmen (9/13)

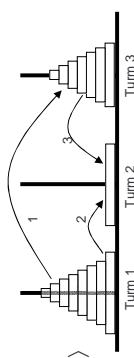
Beispiel 2 : Türme von Hanoi (3/5)

Lösungsansatz:

1. bringe die obersten $n-1$ Scheiben von Turm 1 zu Turm 3;
2. bewege die unterste Scheibe von Turm 1 zu Turm 2;
3. bringe die $n-1$ Scheiben von Turm 3 zu Turm 2.

Algorithmenschema (Java-Pseudocode):

```
void hanoi(int n, Turm t1, Turm t2, Turm t3) {
    // Eingabe: die Anzahl der Scheiben und drei Türme
    // „Ausgabe“: Scheiben sind umgeschichtet
    if (n = 1) <bewege Scheibe von t1 nach t2>
    else {
        hanoi(n-1, t1, t3, t2);
        <bewege oberste Scheibe von t1 nach t2>
        hanoi(n-1, t3, t2, t1);
    }
}
```



Technische Universität
Braunschweig

5-54

„Teile und Herrsche“-Alg. (10/13)

Beispiel 2 : Türme von Hanoi (4/5)

Verfeinerung dieser Lösung führt auf eine rekursive Prozedur: die zu lösenden Teilaufgaben sind von der gleichen Art wie die ursprüngliche Aufgabe, wenn auch kleiner.

Der Aufruf `hanoi(n, t1, t2, t3)` erledigt dann das Geschäft ...

... aber in welcher Zeit ?

Komplexität des Hanoi-Programms:

Nach der Legende ist das Ende der Welt erreicht, wenn die Mönche ihre Aufgabe ganz gelöst haben. Wann ist das bei unserem Algorithmus der Fall ?

Anmerkung: hier ist einer der seltenen Fälle, wo es Sinn macht, nach einem Algorithmus *größtmöglicher* Komplexität zu suchen.



Technische Universität
Braunschweig

5-55

„Teile und Herrsche“-Alg. (11/13)

Beispiel 2 : Türme von Hanoi (5/5)

Komplexität des Hanoi-Programms:
Annahmen: es dauert 1 Minute, eine Scheibe umzulegen, die Mönche lösen sich ab und arbeiten 24 Stunden täglich an der Aufgabe - und zwar *jeden* Tag.

Für n Scheiben braucht man dann die Zeit $T(n)$, für die gilt:

$$T(n) = \begin{cases} 1 & \text{für } n = 1 \\ 2T(n-1)+1 & \text{für } n > 1 \end{cases}$$

Dies ist ohne Probleme lösbar: es ist $T(n) = 2^n - 1$

Für $n = 64$ ist $T(64) = 2^{64} - 1 \approx 10^{19}$ Minuten
 $\approx 10^{14}$ Tage $\approx 3 \cdot 10^{11}$ Jahre
(Glück gehabt ;o)



Technische Universität
Braunschweig

5-56



„Teile und Herrsche“-Alg. (12/13)

Komplexität von „Teile und Herrsche“ allgemein (1/2)

Annahmen:

- die Problemgröße ist durch eine natürliche Zahl n gegeben
- $p = 2$: wir zerlegen in zwei gleich große Teilprobleme
- Zerlegen geht mit $O(1)$
- Zusammensetzen geht mit $O(n)$

Analyse

$$T(1) = c_1$$

$$T(n) =$$

Es kann $c_1 = c_{\text{Zerleg}} = c_{\text{ZSAM}} = 1$ angenommen werden, da die Abschätzung nicht vom Wert der Konstanten abhängt.



Technische Universität
Braunschweig

5-57

„Teile und Herrsche“-Alg. (13/13)

Komplexität von „Teile und Herrsche“ allgemein (2/2)

Also ist für $n = 2^m$ bei jeweils perfekter Halbierung der Probleme:

$$\begin{aligned} T(n) &= T(2^m) = n+1+2T(2^{m-1}) \\ &= n+1+2(n/2 + 1 + 2T(2^{m-2})) \\ &= 2n+2^2-1+2^2T(2^{m-2}) \\ &= \dots \end{aligned}$$

$$= m * n + 2^m - 1 + 2^m T(1)$$

$$= m * n + 2^m - 1 + 2^m$$

$$= n * \log_2 n + 2n - 1$$

$$= O(n * \log n)$$

$$T(1) = 1$$

$$T(n) =$$

Hierbei ist $\log_2 n$ der *Logarithmus dualis* von n , d.h. der Logarithmus zur Basis 2, es ist $\log_2 x = \log_{10} x / \log_{10} 2 \approx 3,322 \log_{10} x$.

Die Abschätzung $O(n \log n)$ gilt auch für $n \neq 2^m$: *TuH* ist also unter den gemachten Annahmen "praktisch so gut wie in linearer Zeit" machbar.

Ein Beispiel ist Sortieren: *TuH* ist deutlich besser als naive Verfahren; z.B. liegt Sortieren durch Einfügen in $\Theta(n^2)$, Merge-Sort hingegen in $\Theta(n \log n)$ [Worst-Case]



Technische Universität
Braunschweig

5-58

„Teile und Herrsche“-Alg. (12/13)

Komplexität von „Teile und Herrsche“ allgemein (1/2)

Annahmen:

- die Problemgröße ist durch eine natürliche Zahl n gegeben
- $p = 2$: wir zerlegen in zwei gleich große Teilprobleme
- Zerlegen geht mit $O(1)$
- Zusammensetzen geht mit $O(n)$

Analyse

$$T(1) = c_1$$

$$T(n) =$$

Es kann $c_1 = c_{\text{Zerleg}} = c_{\text{ZSAM}} = 1$ angenommen werden, da die Abschätzung nicht vom Wert der Konstanten abhängt.



Technische Universität
Braunschweig

5-57

Backtracking-Algorithmen (1/14)

Ziel: Lösungen für Such- und Optimierungsprobleme finden in einer i.a. großen Menge von Möglichkeiten
z.B. *Gewinnstrategie(n) für ein Spiel*

Methode:

- Systematisches Durchsuchen aller Möglichkeiten.
- Geht es auf einem Weg nicht weiter: zurück und einen anderen Weg probieren!

Beispiel 1: Wie kommt die Maus zum Sandwich?

Idee: Maus sucht Labyrinth systematisch ab, d.h.

- Trifft sie das erste Mal auf eine Kreuzung, wählt sie einen beliebig (weiterführenden) Weg.
- Sie merkt sich besuchte Kreuzungen und die eingeschlagene(n) Weg(e).
- Trifft sie auf eine Sackgasse, geht sie zum ersten Kreuzungspunkt zurück, der einen bislang noch nicht eingeschlagenen Weg beinhaltet.



Technische Universität
Braunschweig

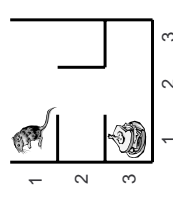
5-59

Backtracking-Algorithmen (2/14)

Beispiel 1 (Fortsetzung): Wie kommt die Maus zum Sandwich?

Modellierung der Weg- und Kreuzungsfolge:

- Wegpunkte als Tupel (x,y) der Raumkoordinaten.
Startpunkt: $(1,1)$, Zielpunkt $(1,3)$
- Bereits besuchte Kreuzungspunkte werden als Knoten eines Baumes aufgefasst, die bereits besuchten Wege als verbindende Kanten.
Startpunkt = Wurzelknoten, Zielpunkt und Sackgassen bilden Blätter



- Der Baum wird nun beginnend mit der Wurzel systematisch aufgebaut:
 1. Einer der möglichen Wege und Kreuzungspunkte wird in den Baum eingetragen und der Weg verfolgt.
 2. Trifft man auf eine Sackgasse, wird der Baum in Richtung der Wurzel nach einem Knoten durchsucht, der noch nicht durchsuchte Wege aufweist (>Weiter mit Schritt 1).
 3. Das Ziel ist erreicht, wenn das Sandwich gefunden ist.



Technische Universität
Braunschweig

5-60

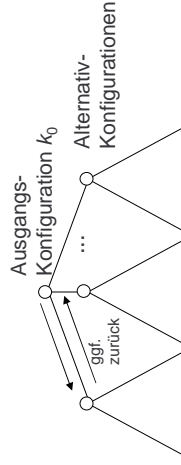
Backtracking-Algorithmen (3/14)

Problemstellung:

- es gibt eine endliche Menge K von Konfigurationen (=: Lösungsraum)
- K ist hierarchisch strukturiert:
 - es gibt eine *Ausgangs-Konfiguration* $k_0 \in K$
 - zu jeder Konfiguration $k_x \in K$ gibt es eine Folge k_{x-1}, \dots, k_{n_x} von direkt erreichbaren *Alternativ-Konfigurationen* (=: *direkte Erweiterungen* von k_x)
- für jede Konfiguration ist entscheidbar, ob sie eine *Lösung* ist
- gesucht werden Lösungen, die von k_0 aus *erreichbar* sind

Strategie:

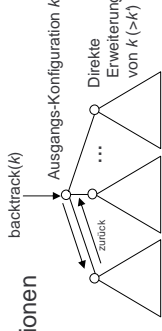
"Sackgassen", (z.B. *hoffnungslose Züge beim Schachspiel*) frühzeitig erkennen und so den Suchaufwand vermindern.



Backtracking-Algorithmen (4/14)

Backtracking-Algorithmenschema:

```
void backtrack(Konfiguration k) {
  // Eingabe: eine Ausgangs-Konfiguration
  // (Virtuelle) Ausgabe: alle Lösungskonfigurationen
  if ( istLösung(k) ) <Gib k aus >;
  else {
    for( (jede direkte Erweiterung k' von k)
         backtrack(k'); // Rekursion !!
    )
  }
}
```







Anmerkungen:

- Terminierung nur dann, wenn Lösungsraum endlich und wiederholtes Betreten einer bereits getesteten Konfiguration auf einem Weg ausgeschlossen (keine Zyklen)
- Ggf. kritisch ist der Aufwand des Backtracking: er hängt von der Größe des Lösungsraumes ab ($O(|K|)$) und ist oft exponentiell (\Rightarrow Varianten).



Backtracking-Algorithmen (5/14)

Varianten des Backtracking:

- Es werden alle bewerteten Lösungen ausgerechnet. Zuletzt wird *die beste ausgewählt* ($>$ Optimierungsprobleme: „finde das größte Sandwich“). 
- Das angegebene Schema findet alle Lösungen. Oft genügt es, eine *erste Lösung* zu finden und den Algorithmus dann zu beenden. 
- Das angegebene Schema besucht jeden „Konfigurationsteilbaum“. Oft kann im voraus entschieden werden, *dass es nicht lohnt, einen solchen Teilbaum zu besuchen* (z.B. wg. Sackgasse ohne Lösung). In diesem Fall kann auf die Bearbeitung des Teilbaums verzichtet werden. Diese Variante wird als „*Branch-and-Bound*“ (abgek. BaB) bezeichnet und bewirkt eine Reduzierung der Komplexität. 
- Aus Komplexitätsgründen wird eine *maximale Rekursionstiefe* vorgegeben, um unter Zeitbeschränkungen wenigstens einen Teil des Lösungsraums zu durchsuchen (als Lösung dient dann z.B. die am besten bewertete zu diesem Zeitpunkt gefundene Lösung). 

Beispiel: Schachprogramm.



Backtracking-Algorithmen (6/14)

„Branch-and-Bound“-Algorithmenschema:

```
void branchAndBound(Konfiguration k) {
  // Eingabe: eine Ausgangs-Konfiguration
  // (Virtuelle) Ausgabe: alle Lösungskonfigurationen
  if ( istLösung(k) ) <Gib k auf Ausgabemedium aus >;
  else { for( (jede direkte Erweiterung k' von k)
              if ( (Lösungen ausgehend von k' überhaupt möglich)
                  branchAndBound(k'); // Rekursion !!
            )
  }
}
```

Anwendungsgebiete des Backtracking bzw. Branch-and-Bound:

- Spiele (Schach, Dame etc.): Konfiguration $k \equiv$ aktuelle Stellung der Figuren, *direkte Erweiterung* \equiv mögliche Spielzüge
- Erfüllbarkeitstests von logischen Aussagen (SAT) und Auswertung von Programmen logischer Programmiersprachen
- Planungsprobleme, Konfigurierungen und Optimierungsprobleme



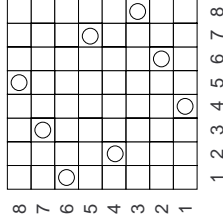
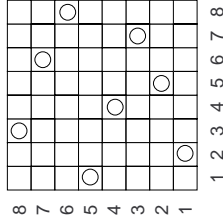
Backtracking-Algorithmen (7/14)

Beispiel 2: das n -Damen-Problem (C.F. Gauß 1850)

Problem:

Es sollen alle Konfigurationen von n Damen auf einem $n \times n$ -Schachbrett erzeugt werden, bei denen keine Dame eine andere "bedroht": keine zwei Damen dürfen in der gleichen Zeile, Spalte oder Diagonale stehen.

Zwei Lösungen (n=8):



Technische Universität
Braunschweig

5-65

Backtracking-Algorithmen (8/14)

Beispiel 2: das n -Damen-Problem (Fortsetzung)

Festlegung des Lösungsraums:

Es ist nicht offensichtlich, wie man die Menge der Konfigurationen K am besten festlegt: alle Belegungen mit $0..n^2$ Damen? ... mit n Damen? ... mit x Damen, die sich nicht gegenseitig bedrohen? Alle diese Mengen sind SEHR groß. Wir treffen folgende klügere Wahl:

K = alle Konfigurationen mit einer Dame in jeder der ersten m Zeilen, $0 \leq m \leq n$, so dass je zwei Damen sich nicht bedrohen.

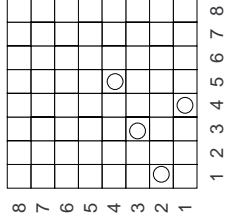
Anmerkungen:

- Wegen der Spiegelsymmetrien kann man K verkleinern (z.B. braucht man für die Dame in der ersten Zeile nur die Spalten 1-4 zu versuchen.)
- K enthält alle Lösungen. Nicht jede Konfiguration lässt sich allerdings zu einer Lösung erweitern (s.z.B. rechts: jedes Feld in Zeile 7 ist bereits bedroht, so dass keine Dame mehr gesetzt werden kann >BaB).



Technische Universität
Braunschweig

5-66

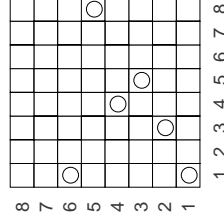


Backtracking-Algorithmen (9/14)

Beispiel 2: das n -Damen-Problem (Fortsetzung)

Anpassung des Backtracking-Algorithmenschemas:

```
void platziere(int zeile) { // Aufruf: platziere(1)
// Eingabe: Nummer der betrachteten Zeile zeile ∈ {1,...,n}
// (Virtuelle) Ausgabe: alle Lösungskonfigurationen
for (int spalte = 1; spalte ≤ n; spalte++)
if ((Feld in Zeile zeile und Spalte spalte nicht bedroht)) {
// Nur sinnvolle Konfigurationen werden betrachtet
<Setze Dame auf Feld (zeile, spalte)>
if ( zeile = n ) <Gib Konfiguration aus>
// Brett voll > Lösung gefunden
else platziere(zeile+1);
// betrachtete direkte
// Erweiterungen der Konfiguration
}
}
```



Technische Universität
Braunschweig

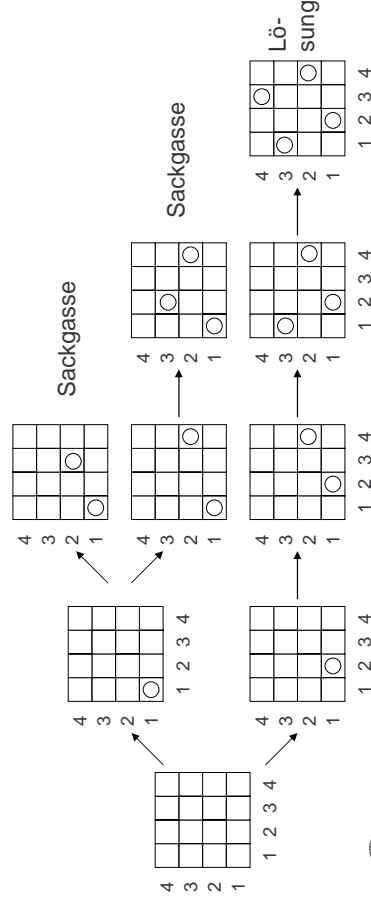
5-67

Backtracking-Algorithmen (10/14)

Beispiel 2: das n -Damen-Problem (Fortsetzung)

Ausführung mit dem 4-Damen-Problem (ohne Symmetrien):

```
void platziere(int zeile) {
for (int spalte = 1; spalte ≤ n; spalte++)
if ((Feld (zeile, spalte) nicht bedroht)) {
<Setze Dame auf Feld (zeile, spalte)>
if ( zeile = n ) <gib Konfiguration aus>
else platziere(zeile+1); } }
```



Technische Universität
Braunschweig

5-68

Backtracking-Algorithmen (1/1/4)

Komplexität des „n-Damen-Problems“

Das n -Damen-Problem ist für $n \geq 4$ lösbar. Wenn die erste Dame nicht richtig gesetzt ist, werden allerdings bis zu $(n-1)!$ Schritte benötigt, um dies herauszufinden. Nach der Stirling'schen Formel ist

der Aufwand also exponentiell! Man kommt auch mit sehr schnellen Rechnern kaum über $n = 100$.

Für jedes $n \geq 4$ ist jedoch ein Verfahren bekannt (Ahrens 1912), wie man in *linearer* (l) Zeit eine Lösung bekommt-allerdings nur eine, nicht alle. Es basiert auf der Beobachtung, dass in Lösungsmustern häufig Rösselsprung-Sequenzen auftreten. Hier ist eine Lösung für $n=6$:

6									
5									
4									
3									
2									
1									
	1	2	3	4	5	6			

5-69



Backtracking-Algorithmen (12/14)

Beispiel 3: das Problem des Handlungsreisenden (Traveling Salesman Problem: TSP)

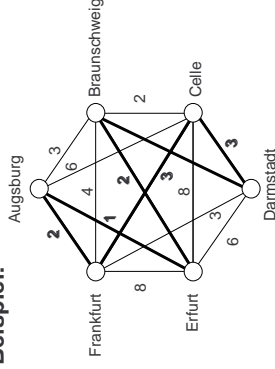
Gegeben: n durch Straßen verbundene Städte mit Reisekosten $c(i, j)$ zwischen je zwei¹ Städten i und j

Gesucht: billigste Rundreise, die jede Stadt genau einmal besucht².

Beispiel: Die billigste Rundreise kostet 11 Einheiten.

Fußnoten:

1. Dies ist ein *vollständiger Graph*: je zwei Knoten sind durch eine Kante verbunden. Ist allerdings $c(i, j)$ sehr groß, so wird die Kante nie gewählt und kann ebenso gut von vornherein fehlen.
2. In der Graphentheorie ist eine solche Kantenfolge als *Hamiltonscher Zyklus* bekannt.



5-70

Backtracking-Algorithmen (13/14)

Komplexität des „n-Damen-Problems“

Das n -Damen-Problem ist für $n \geq 4$ lösbar. Wenn die erste Dame nicht richtig gesetzt ist, werden allerdings bis zu $(n-1)!$ Schritte benötigt, um dies herauszufinden. Nach der Stirling'schen Formel ist

der Aufwand also exponentiell! Man kommt auch mit sehr schnellen Rechnern kaum über $n = 100$.

Für jedes $n \geq 4$ ist jedoch ein Verfahren bekannt (Ahrens 1912), wie man in *linearer* (l) Zeit eine Lösung bekommt-allerdings nur eine, nicht alle. Es basiert auf der Beobachtung, dass in Lösungsmustern häufig Rösselsprung-Sequenzen auftreten. Hier ist eine Lösung für $n=6$:

6									
5									
4									
3									
2									
1									
	1	2	3	4	5	6			

5-69



Backtracking-Algorithmen (13/14)

Beispiel 3: TSP (Fortsetzung), Lösungen (1/2)

Backtracking: In der naiven Form werden alle Wege abgesucht, ausgehend von einem Startknoten.

Komplexität: $O(n!)$, denn vom Startknoten sind n Kanten zu verfolgen, vom nächsten Knoten $(n-1)$, dann $(n-2)$ u.s.w.

Es ist $O(n!) > O(2^n)$, und das ist wirklich *sehr* viel. Naives Backtracking ist unpraktikabel bei Problemen oberhalb der Größenordnung, die man *fast* noch durch scharfes Hinsehen lösen kann.

Greedy: Folgendes Verfahren führt zu einer Näherungslösung:

1. sortiere Kanten nach Kosten $\Rightarrow O(n^2 \log n^2)$,
2. wähle billigste Kante unter den Nebenbedingungen
 - es darf kein Zyklus entstehen - außer am Schluss die Rundreise,
 - kein Knoten darf mehr als 2 Kanten haben - das ist bei *jeder Rundreise* so.



5-71

Backtracking-Algorithmen (14/14)

Beispiel 3: TSP (Fortsetzung), Lösungen

Greedy (Fortsetzung):

Komplexität: Dies Greedy-Verfahren ist schnell, nämlich $O(n^2 \log n^2)$.

Aber es führt nicht immer zu einer globalen Lösung. Trotzdem wird es in der Praxis erfolgreich eingesetzt: die Näherungslösungen sind gar nicht so schlecht.

Branch-and-Bound: Wenn man weiß, dass eine Lösung mit Kosten k existiert (z.B. durch den obigen Greedy-Algorithmus¹), dann kann man beim Backtracking alle Teillösungen abschneiden, die bereits teurer sind.

Komplexität: Eine allgemeine Abschätzung ist unmöglich, es hängt davon ab, wie gute Schranken man finden kann. Aber nur unter *sehr* glücklichen Umständen kommt man unter $O(2^n)$. Immerhin ist das viel besser als $O(n!)$, aber praktikabel wird es in der Regel nicht.

¹ Man kann u.a. auch $k=2 \cdot m$ nehmen, wenn m die Kosten eines minimalen spannenden Baums sind. Warum?



5-72

7.3 Konstruktionsprinzipien

Schrittweise Verfeinerung (1/3)

... ist eine weithin empfohlene und bewährte Vorgehensweise beim Entwurf und der Programmierung von Algorithmen.

Grundidee :

1. entwerfe einen *abstrakten Algorithmus* auf *abstrakten Daten*
2. konkretisiere dies durch *schrittweise Verfeinerung*

Entscheidungen über konkrete Darstellungen - insbesondere der Daten - durch Mittel der Programmiersprache sind *möglichst spät* zu treffen.

Denn um Entscheidungen über *effiziente Realisierungen* fundiert treffen zu können, sollte man möglichst viel über die Details des Algorithmus wissen, z.B. welche Operationen (am häufigsten) auf welchen Daten ausgeführt werden. Eine zu frühe Entscheidung für eine vermeintlich effiziente Realisierung kann kontraproduktiv sein, d.h. die Sicht auf bessere Lösungen verbauen.

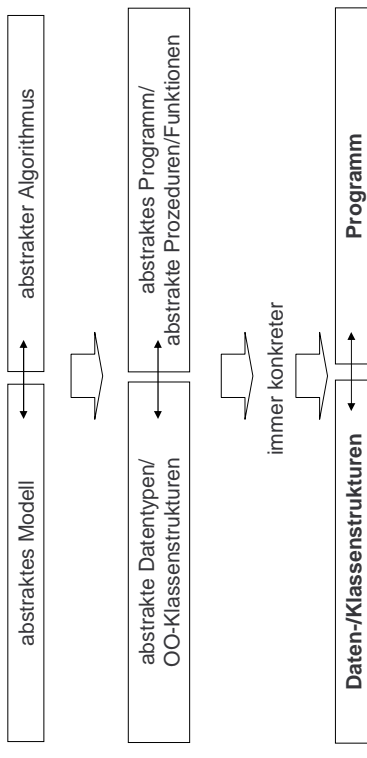


Technische Universität
Braunschweig

5-73

Schrittweise Verfeinerung (2/3)

Schema der Schrittweisen Verfeinerung



↔ : Wechselseitige Beeinflussung
Grenzen sind fließend



Technische Universität
Braunschweig

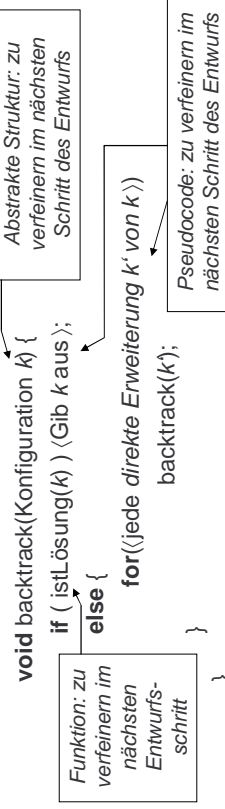
5-74

Schrittweise Verfeinerung (3/4)

Hilfsmittel der schrittweisen Verfeinerung (bereits verwendet):

- Erste Formulierung in Pseudocode
- Aufteilen eines Problems in Teilprobleme, die von entsprechenden Prozeduren/Funktionen zu lösen sind. Letztere können dann in einem weiteren Schritt ausformuliert werden, wobei dieses Verfahren rekursiv angewendet werden kann: erneute Aufteilung in Teilprobleme, Erledigung durch geeignete Prozeduren/Funktionen/elementare Anweisungen etc. Die meisten Programmiersprachen unterstützen dieses Verfahren direkt.

Beispiel (s.o.):



Technische Universität
Braunschweig

5-75

Schrittweise Verfeinerung (4/4)

Schrittweise Verfeinerung (3/3)

Schema der Schrittweisen Verfeinerung als (Meta-)Algorithmus (Achtung: hier ist der/die EntwicklerIn der/die Ausführende)

```
void schrittweiseVerfeinerung(Problem P) {
    if ( klein(P) ) expliziteLösungDurchElementareAnweisungen( P )
    else {
```

```
        Problem[] TP = teileAuf( P ); // Array der p Teilproblem
```

```
        for ( i=1; k=p; i++) {
```

```
            <Schreibe Prozedur-/Funktionsaufruf & -deklaration, der/die TP[i] löst>
            schrittweiseVerfeinerung(TP[i])
        }
```

```
    }
```

Anmerkung: Entscheidung, ob TP durch elementare Anweisungen direkt, oder durch eine Prozedur/Funktion gelöst wird, ist abhängig von Effizienz-, Wartbarkeits- und Wiederverwendbarkeitsüberlegungen



Technische Universität
Braunschweig

5-76

Einsatz von Algorithmenmustern

Grundidee: Wiederverwendung

Anwendungsschema:

1. Erkenne Problemmuster in dem konkret zu lösenden Problem wieder
2. Passe Algorithmenmuster und ggf. Datenstrukturen an konkretes Problem an (bzw. vice versa).

Bisher behandelte Algorithmenmuster:

Greedy-Algorithmen, "Teile-und-Herrsche"-Algorithmen, Backtracking
Weitere Algorithmenmuster: Dynamische Programmierung (AuD II), ...

Weitere Mustertypen in der Informatik

- **Analysemuster:** I.d.R. OO-Klassenstrukturen (ggf. inkl. Verhalten) zur Beschreibung eines oft wiederkehrenden Problembereichs
- **Entwurfsmuster:** I.d.R. OO-Klassenstrukturen (i.d.R. inkl. Verhalten) zur Beschreibung einer *Standardlösung* zu einem oft wiederkehrenden Problem(bereich). Oft auch Bestandteil einer Programmiersprache (Bibliotheken, parametrierbare Algorithmen).



Technische Universität
Braunschweig

5-77

Problemreduzierung durch Rekursion

Grundidee:

- Wiederholtes Anwenden des gleichen Lösungsschemas auf ein (im Ausgangsproblem enthaltenes) immer kleiner werdendes Teilproblem.
- Abbruch der Wiederholung, wenn ein zu lösendes Teilproblem so "klein" ist, dass es direkt gelöst werden kann.

Anmerkungen:

1. Rekursion bietet sich an, wenn die Problemstruktur rekursiv aufgebaut ist.
2. Zu rekursiven Lösungen gibt es immer auch iterative Entsprechungen. Bei der Entscheidung muss die Effizienz des resultierenden Programms und der Programmierstellung berücksichtigt werden (s. Kap. 4).
3. Rekursion ist ein für die Informatik spezifischer Lösungsansatz, der in den klassischen Ingenieurwissenschaften nicht vorkommt. Er ist nicht aus dem Alltagswissen ableitbar und muss erlernt und geübt werden.
4. Rekursion ist in den vorangegangenen Kapiteln häufig angewendet worden und findet sich auch in Algorithmenmustern wieder.



Technische Universität
Braunschweig

5-78

7.4 Verifikation

Es ist manchmal nötig, die *Korrektheit* von Computersystemen (Hardware oder Software oder beides in Kombination) nachzuweisen. Konventionelles Testen ist dann nicht ausreichend.

Besonders in folgenden Fällen: das System ist ...

- **sicherheitskritisch:** das Versagen des Systems gefährdet Gesundheit und Leben, z.B.: Medizin, Verkehr, Raumfahrt, Prozesssteuerung in technischen Anlagen (Chemiewerk, Reaktor, ...), ...
- **kommerziell kritisch:** das Versagen des Systems gefährdet kommerzielle Unternehmungen, z.B.: massenproduzierte Chips bzw. Standardsoftware, Datenverwaltung (Datenverlust!), Anwendungen in Banken, Börsen, ...
- **politisch kritisch:** das Versagen des Systems gefährdet politisch wichtige Belange, z.B.: Raumfahrtprojekte von nationaler Bedeutung, ...



Technische Universität
Braunschweig

5-79

Probleme von Verifikationsmethoden

Verifikationsmethoden werden oft als unpraktikabel abgetan, da gerade komplexe Programmsysteme (und die sind heutzutage die Regel) bislang nur mit großem Aufwand formal zu verifizieren sind.

Dennoch finden die Methoden immer mehr Verwendung, in verschiedenen Anwendungsbereichen (s.o.) geht auch kein Weg daran vorbei. Sie sind aber noch nicht ausgereift. Einer der Gründe: es fehlt an wissenschaftlichen Grundlagen. *Der Bedarf an Grundlagenforschung ist groß.*

Erforderlich sind Methoden, Sprachen und Werkzeuge zur

- **Modellierung** von Systemen auf hoher Abstraktionsebene
- **(Formalen) Spezifikation** nachzuweisender Eigenschaften dieser Systeme (Terminierungsverhalten, berechnete Funktionswerte etc.)
- **Verifikation**, d.h. zum formalen Beweis, dass ein implementiertes System die spezifizierten Eigenschaften hat
(Achtung: Validierung = (nichtformaler) Nachweis der Übereinstimmung mit einer (oft informellen) Spezifikation i.d.R. durch systematisches Testen)



Technische Universität
Braunschweig

5-80

Ansätze und Aspekte (1/2)

Formale Beweise

Das Programm wird als Menge von Formeln ϕ in einer geeigneten Logik modelliert, und die Spezifikation als Formel φ in dieser Logik.

Dann wird mit Beweisregeln der formalen Logik gezeigt, dass ϕ aus φ hergeleitet werden kann: $\phi \vdash \varphi$.



Model Checking

Das Programm wird als Modell M in einer geeigneten Logik modelliert, und die Spezifikation wieder als Formel φ in dieser Logik.

Dann wird nachgeprüft, ob φ im Modell M gilt: $\models_M \varphi$



Automatisierung

Die Ansätze unterscheiden sich darin, welchen Grad von Automatisierung sie aufweisen bzw. zulassen. Extreme sind voll automatisch und voll manuell, aber es gibt eine Reihe von CA (computer assisted)-Methoden dazwischen.



Technische Universität
Braunschweig

5-87

Ansätze und Aspekte (2/2)

Vollständigkeit

Die Spezifikation kann einzelne Eigenschaften eines Systems beschreiben oder dessen Verhalten vollständig spezifizieren. Letzteres ist typischerweise sehr teuer zu verifizieren.

Anwendungsbereich

Handelt es sich um Hardware oder Software oder beides in Kombination? Um ein terminierendes Programm, das eine Funktion berechnet? Oder um ein reaktives System, das niemals anhält? Kann man spezifisches Wissen aus dem Anwendungsbereich ausnutzen? ...

Zeitpunkt

Verifikation ist vorteilhafter, wenn man sie *frühzeitig* i.d. *Softwareentwicklung* anwendet: je eher ein Fehler erkannt wird, um so billiger ist er zu korrigieren.

Die nachfolgend behandelte Methode betrifft formale Beweise, ist allenfalls halbautomatisch durchführbar, i.a. nicht vollständig und betrifft terminierende Programme, die Anfangs- in Endzustände überführen. Sie kann also erst relativ spät im Softwareentwicklungsprozess eingesetzt werden.



Technische Universität
Braunschweig

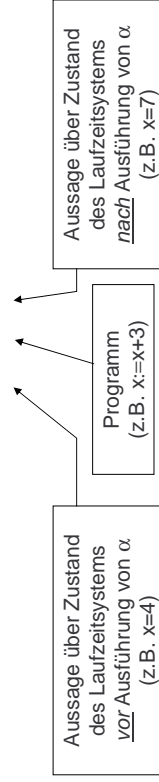
5-82

Korrektheit (anschaulich) (1/2)

Im Folgenden wird die Korrektheit imperativer Programme untersucht. **Aber was heißt Korrektheit?**

Zunächst hat die Spezifikation gewünschter Eigenschaften eines Programms α zu erfolgen: im Folgenden sei sie durch Angabe einer (zustandsbezogenen) *Vorbedingung* P und einer (dito) *Nachbedingung* Q zum Programm α gegeben.

„Syntax“ dieser sog. *Formeln*: $\{ P \} \alpha \{ Q \}$



Bedeutung: wenn die Vorbedingung P vor Ausführung von α gilt, so gilt die Nachbedingung Q nach Ausführung von α (exakter: genau dann, wenn letzteres beweisbar der Fall ist *gilt* die Formel).



Technische Universität
Braunschweig

5-83

Korrektheit (anschaulich) (2/2)

Fortsetzung: Was heißt Korrektheit?

- Ein Programm α heißt *partiell korrekt* bezüglich P und Q gdw. $\{ P \} \alpha \{ Q \}$ *formal beweisbar gilt* (d.h. "wahr" ist).

Diese Definition verlangt *nicht*, dass α *terminiert*, damit die partielle Korrektheit gilt. Terminiert α nicht, so ist die Nachbedingung irrelevant. Aus diesem Grund wird der Begriff der *totalen Korrektheit* eingeführt:

- Ein Programm α heißt *total korrekt* bezüglich P und Q gdw. α *partiell korrekt ist und zudem immer dann terminiert*, wenn Vorbedingung P gilt.

Partielle Korrektheit: falls dieser Punkt erreicht wird, dann ist dies richtig gem. Q



Totale Korrektheit: dieser Punkt wird erreicht und dann ist dies richtig gem. Q



Technische Universität
Braunschweig

5-84

Basissprache(n) (1/2)

Die Methode bedarf der Formalisierung und wird anhand von drei aufeinander bezogenen "Spielzeug"-Sprachen demonstriert. Syntax:

- E (expressions): Ausdrücke, hier nur vom Typ *int*
 - B (boolean expressions, Bedingungen): Formeln zur Formulierung von Aussagen über Programmzustände
 - C (commands): Anweisungen einer einfachen iterativen Programmiersprache
- $$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E * E) \mid (E \div E)$$
- n steht für eine beliebige ganze Zahl, x für eine beliebige Variable vom Typ *int*.
Wir machen Gebrauch von den üblichen Regeln zur Einsparung von Klammern.
- $$B ::= true \mid (-B) \mid (B \wedge B) \mid (B \vee B) \mid (E < E)$$
- weitere Formeln können damit definiert werden, z.B. *false* als $(-true)$, $(B_1 \Rightarrow B_2)$ als $(-B_1 \vee B_2)$ oder $(E_1 = E_2)$ als $(-(E_1 < E_2) \wedge -(E_2 < E_1))$.
- $$C ::= x := E \mid C; C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \text{ od}$$
- dies sind die bereits eingeführten Anweisungen.

Basissprache(n) (2/2)

Beispiele:

- *Zuweisungen:*
 1. $x := 1$
 2. $x := x+1$
 3. $x := y$
 4. $z := x; x := y; y := z$
- *Bedingte Anweisungen:*
 1. if even(x) then $x := x + 2$
 else $x := x-1$ fi
- *Schleifen:*
 - while *true* do $x := 1$ od
 - while $x \neq 0$ do $x := x-1$ od
 - while $p(x)$ do $x := x-1$ od // $p(x) = \text{bet. Bootescher Term mit } x$
 - while $x \neq 0$ do $x := x-1; y := y+1$ od

Wichtig: Festlegung der Syntax

Erinnerung:

die Syntax der Anweisungen (Commands) etc. war in Kapitel 4 *denotational* festgelegt worden (sog. *Funktionensemantik*), z.B. Semantikfestlegung der *Zuweisung*:

$$\llbracket x := v \rrbracket (\sigma) = \sigma_{(x \leftarrow v)}$$

Axiomatische Semantikfestlegung

Im Folgenden erfolgt die Semantikfestlegung „axiomatisch“, z.B. *Zuweisung*:

- $\{\varphi\} x := t \{\psi\}$ gilt, wenn in φ jedes Vorkommen von Term t (rechte Seite) durch x (linke Seite) ersetzt wird, wodurch ψ entsteht.
- **Beispiel:** $\{x + 1 = 2\} x := x+1 \{x = 2\}$ gilt offensichtlich;
Anwendung obiger Regel:
 $\varphi \equiv x+1=2 \rightarrow "x+1"$ durch " x " (linke Seite) ersetzen $\rightarrow x=2 \equiv \varphi'$

Durch die *axiomatische Semantikspezifikation* entsteht die Möglichkeit, formale Beweise über Eigenschaften von Programmen zu führen.

Außerdem lassen sich nach obigem Muster für die Zuweisung sogenannte *Beweisregeln* für alle Anweisungstypen angeben und bei Beweisen von Eigenschaften komplexer Programme benutzen.

Anmerkung: Neben der *denotationalen* und der *axiomatischen* Semantikfestlegung ist eine dritte zu erwähnen: die *operationale* Semantikfestlegung (s.z.B. Turing-Maschinen). Bei letzterer wird ein sog. *Interpreter* für eine *Programmiersprache* angegeben, der aus *Eingabedaten* durch *schriftweise Abarbeitung* des Programms *Ausgabedaten* erzeugt.

Hoare-Logik (1/2)

Programmtexte mit Kommentaren {...} werden zu einer formalen Programmlogik präzisiert. Deren Formeln heißen auch „Hoare-Tripel“

- **Syntax:** $\{ P \} \alpha \{ Q \}$ (s.o.)
- $\{ P \} \alpha \{ Q \}$ heißen *Formeln* mit $P, Q \in B, \alpha \in C$
 P, Q heißen auch *Zusicherungen*
- **Korrektheit** bzw. Gültigkeit, zwei Varianten (informell):
partiell $\models_p \{ P \} \alpha \{ Q \}$
wenn (P gilt vor α **und** α terminiert), **dann** (Q gilt nach α)
- **total** $\models_t \{ P \} \alpha \{ Q \}$
wenn (P gilt vor α), **dann** (α terminiert **und** Q gilt nach α)
- **Notation:**
wir beschäftigen uns überwiegend mit partieller Korrektheit, daher lassen wird " \models_p " auch weg: " $\{ P \} \alpha \{ Q \}$ " steht für " $\models_p \{ P \} \alpha \{ Q \}$ ".

Anmerkung: Von C.A.R. Hoare 1996 eingeführt.

Hoare-Logik (2/2)

Beispiele: folgende Aussagen über partielle Korrektheit gelten:

- $\{true\} x := 1 \{x = 1\}$
- $\{x = 1\} x := x+1 \{x = 2\}$
- $\{y = a\} x := y \{y = a \wedge x = a\}$
- $\{x = a \wedge y = b\} z := x; x := y; y := z \{x = b \wedge y = a\}$
- $\{false\} x := 1 \{x = 2\}$
- $\{true\} \text{while } true \text{ do } x := 1 \text{ od } \{x = 2\}$
- $\{x > 0\} \text{while } x \neq 0 \text{ do } x := x-1 \text{ od } \{x = 0\}$
- $\{true\} \text{while } x \neq 0 \text{ do } x := x-1 \text{ od } \{x = 0\}$
- $\{true\} \text{while } p(x) \text{ do } \alpha \text{ od } \{\neg p(x)\}$
- $\{x+y = a\} \text{while } x \neq 0 \text{ do } x := x-1; y := y+1 \text{ od } \{x = 0 \wedge x+y = a\}$



Technische Universität
Braunschweig

5-89

Erinnerung Implikation
(\Rightarrow ; „daraus folgt“)

p	q	$p \Rightarrow q$
false	false	true (!)
false	true	true (!)
true	false	false
true	true	true

Anschaulich:
eine Aussage

$\{P\} \alpha \{Q\}$

gilt also genau dann,
wenn

$P \Rightarrow \dots \Rightarrow Q$

beweisbar „wahr bzw.
true“ ist.



Technische Universität
Braunschweig

5-91

Beweisregeln f. part. Korrektheit (1/7)

Beweisregeln erlauben es, neue Formeln aus gegebenen Mengen von Formeln syntaktisch herzuleiten:

Sei ϕ eine Formel, und sei Φ eine Menge von Formeln.

$\Phi \vdash \phi$ oder $\Phi \vdash \phi$ bedeutet, dass ϕ aus Φ herzuleiten ist.

Notationsmuster $\frac{\{\phi_{11}\} \alpha_1 \quad \{\phi_{12}\}, \{\phi_{21}\} \alpha_2 \quad \{\phi_{22}\}, \dots, \{\phi_{n1}\} \alpha_n \quad \{\phi_{n2}\}}{\text{(i.d.R.)} \quad \Phi} = \Phi$
 $\{\psi_1\} \beta \quad \{\psi_2\}$

Bedeutung: Wenn $\{\phi_{11}\} \alpha_1 \quad \{\phi_{12}\}$ und $\{\phi_{21}\} \alpha_2 \quad \{\phi_{22}\}$ und... und $\{\phi_{n1}\} \alpha_n \quad \{\phi_{n2}\}$ gilt, dann auch (ohne weiteren Beweis) $\{\psi_1\} \beta \quad \{\psi_2\}$.

Dabei darf Φ auch die leere Formel sein. In diesem Fall gibt es keine weitere Voraussetzung für die Anwendung der Regel.

I.d.R. ist $\phi_{11} = \psi_1$ und $\phi_{n2} = \psi_2$.

Bei den Beweisregeln für die Hoare-Logik unserer einfachen Programmiersprache lassen wir die Mengenklammern "{'" bzw. "}" bei Φ üblicherweise weg.



Technische Universität
Braunschweig

5-90

Beweisregeln f. part. Korrektheit (2/7)

1. Zuweisung

$\{\phi[t/x]\} x := t \{\phi\}$

Leere Formel Φ , d.h. keine weitere Voraussetzung für die Anwendung der Regel.

Anwendung: in $\phi[t/x]$ kann jedes Vorkommen des Terms t (rechte Seite) durch 'x' (linke Seite) ersetzt werden, wodurch ϕ entsteht.

Beispiele von Regelanwendungen zur Konstruktion gültiger Formeln:

- s. oben
- $\{-\{2 = 2\} x := 2 \{x = 2\}$
// "2" \equiv t wird durch
// "x" \equiv x ersetzt
- $\{-\{x+1 = y\} x := x+1 \{x = y\}$
- $\{-\{x+1+5 = y\} x := x+1 \{x+5 = y\}$
// Hier wurde $\phi[t/x] = "x+6"$ vorbereitend um-
// geformt (um t zu "erzeugen") > Anpassung
- $\{-\{x+1 > 0 \wedge y > 0\} x := x+1 \{x > 0 \wedge y > 0\}$
// $\phi[t/x] =$ darf beliebig komplexer logischer
// Ausdruck sein



Technische Universität
Braunschweig

5-91

Beweisregeln f. part. Korrektheit (3/7)

2. Anpassungsregel (Hilfsregel)

$\frac{\phi_1 \Rightarrow \phi_2, \{\phi_2\} \alpha \quad \{\phi_3\}, \phi_3 \Rightarrow \phi_4}{\{\phi_1\} \alpha \quad \{\phi_4\}}$

Kurzform:

$\frac{\{\phi_1\} \Rightarrow \{\phi_2\} \quad \alpha \quad \{\phi_3\} \Rightarrow \{\phi_4\}}{\{\phi_1\} \alpha \quad \{\phi_4\}}$

Bedeutung: Wenn $\{\phi_2\} \alpha \quad \{\phi_3\}$ gilt und aus ϕ_1, ϕ_2 sowie aus ϕ_3, ϕ_4 logisch abzuleiten sind, dann gilt auch $\{\phi_1\} \alpha \quad \{\phi_4\}$.

Anmerkung: Regel dient zur „Rechtfertigung“ logischer Umformungen der Vor- und Nachbedingungen (i.d.R. zur Vorbereitung der Anwendung anderer Regeln). Wird oft implizit verwendet.

Beispiele:

$x+6 = y \Rightarrow x+1+5 = y, \{x+1+5 = y\} x := x+1 \{x+5 = y\}$
 $\{x+6 = y\} x := x+1 \{x+5 = y\}$

$\{0 < x \wedge x = a \wedge \text{even}(x)\} \Rightarrow \{0 < x+2 \wedge x+2 = a+2 \wedge \text{even}(2x+2)\}$
 $x := x+2 \{0 < x \wedge x = a+2 \wedge \text{even}(2x)\} \Rightarrow \{0 \leq x < a\}$

$\{0 < x \wedge x = a \wedge \text{even}(x)\} x := x+2 \{0 \leq x < a\}$



Technische Universität
Braunschweig

5-92

Kombination von
Anpassungsregel
(nur Vorbedingung)
und
Zuweisungsregel

dito
(Kurzform)

5-92

Beweisregeln f. part. Korrektheit (4/7)

3. Sequenz

$$\frac{\{\varphi_1\} \alpha_1 \{ \varphi_2 \}, \{ \varphi_2 \} \alpha_2 \{ \varphi_3 \}}{\{\varphi_1\} \alpha_1; \alpha_2 \{ \varphi_3 \}}$$
 φ_2 heißt auch Zwischenzusicherung

Bedeutung: wenn $\{\varphi_1\} \alpha_1 \{ \varphi_2 \}$ gilt und $\{\varphi_2\} \alpha_2 \{ \varphi_3 \}$ gilt, dann gilt für die Hintereinanderausführung von α_1 und α_2 : $\{\varphi_1\} \alpha_1; \alpha_2 \{ \varphi_3 \}$.

Beispiel:

$\{y+2 = 3\} x := 2 \{y+x = 3\}$ und $\{y+x = 3\} y := y+x \{y = 3\}$ gelten aufgrund der Beweisregeln für die Zuweisung.

Damit ist für das Programmfragment "x := 2 ; y := y+x" und die Vorbedingung $\{y+2 = 3\}$ folgende Formel mittels oben genannter Beweisregel abzuleiten :

$$\frac{\{y+2 = 3\} x := 2 \{y+x = 3\}, \{y+x = 3\} y := y+x \{y = 3\}}{\{y+2 = 3\} x := 2 ; y := y+x \{y = 3\}}$$


Technische Universität
Braunschweig

5-93

Beweisregeln f. part. Korrektheit (5/7)

4. Bedingte Ausführung (Selektion)

$$\frac{\{\varphi_1 \wedge B\} \alpha \{ \varphi_2 \}, \{ \varphi_1 \wedge \neg B \} \beta \{ \varphi_2 \}}{\{\varphi_1\} \text{if } B \text{ then } \alpha \text{ else } \beta \text{ fi } \{ \varphi_2 \}}$$

Bedeutung: Die Vorbedingung φ_1 muss bei zutreffender Bedingung B durch α zur Nachbedingung φ_2 abzuleiten sein genauso wie φ_1 bei nicht zutreffender Bedingung B durch β zur Nachbedingung φ_2 abzuleiten sein muss. Dann gilt für die bedingte Ausführung von α oder β abhängig von $B \{ \varphi_1 \}$ if B then α else β fi $\{ \varphi_2 \}$

Beispiel: z.Z.: $\{0 < x \wedge x = a\}$ if even(x) then $x := x+2$ else $x := x-1$ fi $\{0 \leq x < a\}$

 $\{0 < x \wedge x = a \wedge \text{even}(x)\} x := x+2 \{0 \leq x < a\},$
 $\{0 < x \wedge x = a \wedge \neg \text{even}(x)\} x := x-1 \{0 \leq x < a\}$

gilt (Zuweisungsregel und Anpassung (wg. even: s.o.))

 $\{0 < x \wedge x = a\} \text{if even}(x) \text{ then } x := x+2 \text{ else } x := x-1 \text{ fi } \{0 \leq x < a\}$

gilt somit auch

5-94



Technische Universität
Braunschweig

Beweisregeln f. part. Korrektheit (6/7)

3. Sequenz

$$\frac{\{\varphi_1\} \alpha_1 \{ \varphi_2 \}, \{ \varphi_2 \} \alpha_2 \{ \varphi_3 \}}{\{\varphi_1\} \alpha_1; \alpha_2 \{ \varphi_3 \}}$$
 φ_2 heißt auch Zwischenzusicherung

Bedeutung: wenn $\{\varphi_1\} \alpha_1 \{ \varphi_2 \}$ gilt und $\{\varphi_2\} \alpha_2 \{ \varphi_3 \}$ gilt, dann gilt für die Hintereinanderausführung von α_1 und α_2 : $\{\varphi_1\} \alpha_1; \alpha_2 \{ \varphi_3 \}$.

Beispiel:

$\{y+2 = 3\} x := 2 \{y+x = 3\}$ und $\{y+x = 3\} y := y+x \{y = 3\}$ gelten aufgrund der Beweisregeln für die Zuweisung.

Damit ist für das Programmfragment "x := 2 ; y := y+x" und die Vorbedingung $\{y+2 = 3\}$ folgende Formel mittels oben genannter Beweisregel abzuleiten :

$$\frac{\{y+2 = 3\} x := 2 \{y+x = 3\}, \{y+x = 3\} y := y+x \{y = 3\}}{\{y+2 = 3\} x := 2 ; y := y+x \{y = 3\}}$$


Technische Universität
Braunschweig

5-93

Beweisregeln f. part. Korrektheit (6/7)

5. Einseitige bedingte Ausführung (Selektion)

$$\frac{\{\varphi_1 \wedge B\} \alpha \{ \varphi_2 \}, (\varphi_1 \wedge \neg B) \Rightarrow \varphi_2}{\{\varphi_1\} \text{if } B \text{ then } \alpha \text{ fi } \{ \varphi_2 \}}$$

Bedeutung: Die Vorbedingung φ_1 muss bei zutreffender Bedingung B durch α zur Nachbedingung φ_2 abzuleiten sein genauso wie φ_1 bei nicht zutreffender Bedingung B direkt zur Nachbedingung φ_2 abzuleiten sein muss. Dann gilt für die bedingte Ausführung von α abhängig von $B \{ \varphi_1 \}$ if B then α fi $\{ \varphi_2 \}$

Beispiel: z.Z.: $\{0 < x \wedge x < a\}$ if even(x) then $x := x+2$ fi $\{0 \leq x < a\}$

 $\{0 < x \wedge x < a \wedge \text{even}(x)\} x := x+2 \{0 \leq x < a\},$
 $0 < x \wedge x < a \wedge \neg \text{even}(x) \Rightarrow 0 \leq x < a$

gilt (Zuweisungsregel und Anpassung (s.o.))

gilt

 $\{0 < x \wedge x < a\} \text{if even}(x) \text{ then } x := x+2 \text{ fi } \{0 \leq x < a\}$

gilt somit auch



Technische Universität
Braunschweig

5-95

Beweisregeln f. part. Korrektheit (7/7)

6. Iteration

$$\frac{\{\varphi \wedge B\} \alpha \{ \varphi \}}{\{\varphi\} \text{while } B \text{ do } \alpha \text{ od } \{ \varphi \wedge \neg B \}}$$

Bedeutung: φ ist eine Schleifeninvariante: φ gilt nach α , sofern φ vorher gilt und die Schleife noch einmal ausgeführt werden soll (Bedingung B). Wird α aufgrund des Nichtzutreffens der Bedingung B nicht mehr ausgeführt (Abbruch der Iteration), muss danach $\varphi \wedge \neg B$ gelten.

Beispiel: z.Z.: $\{0 < x\}$ while $x > 1$ do $x := x \div 2$ od $\{x = 1\}$

 $\{0 < x \wedge x > 1\} x := x \div 2 \{0 < x\}$

gilt (Zuweisungsregel und Anpassung (implizit))

 $\{0 < x\} \text{while } x > 1 \text{ do } x := x \div 2 \text{ od } \{0 < x \wedge \neg x > 1\}$

gilt somit auch ($x=1$ folgt mit Anpassung)

Anmerkung: Schwierig ist vor allem das Finden einer geeigneten Schleifeninvariante!



Technische Universität
Braunschweig

5-96

Abgeleitete Regeln

Aus den obigen Grundregeln lassen sich weitere ableiten. Gebräuchlich sind z.B. die folgenden:

A1. Mehrfachsequenz

$$\frac{\{\varphi_1\} \alpha_1 \{ \varphi_2 \}, \{ \varphi_2 \} \alpha_2 \{ \varphi_3 \}, \dots, \{ \varphi_{n-1} \} \alpha_n \{ \varphi_n \}}{\{\varphi_1\} \alpha_1 ; \alpha_2 ; \dots ; \alpha_n \{ \varphi_n \}}$$

A2. Iteration mit Anpassung

$$\frac{\varphi_1 \Rightarrow \varphi_2, \{ \varphi_2 \wedge B \} \alpha \{ \varphi_2 \}, (\varphi_2 \wedge \neg B) \Rightarrow \varphi_3}{\{\varphi_1\} \text{ while } B \text{ do } \alpha \text{ od } \{ \varphi_3 \}}$$



Technische Universität
Braunschweig

5-97

Schleifeninvariante (1/4)

Schwieriger Punkt bei der Verifikation von Iterationen (Schleifen):

man muss eine Schleifeninvariante φ finden, die vom Innern (Rumpf) der Schleife unverändert gelassen wird, wenn die "Weitermachen"-Bedingung B der Schleife erfüllt ist:

$$\{\varphi \wedge B\} \alpha \{ \varphi \}$$

Iterationsregel:

$$\frac{\{\varphi \wedge B\} \alpha \{ \varphi \}}{\{\varphi\} \text{ while } B \text{ do } \alpha \text{ od } \{ \varphi \wedge \neg B \}}$$

φ gilt dann vor und nach *jedem* Schleifendurchlauf - und somit auch nach dem letzten, d.h. am Ende der Schleife. Dort gilt B dann nicht mehr:

$$\{\varphi\} \text{ while } B \text{ do } \alpha \text{ od } \{ \varphi \wedge \neg B \}$$

Um Schleifenvarianten zu gewinnen, gibt es kein allgemeines Rezept. Es hilft nur: scharf hinschauen, genau verstehen, was vorgeht, und sich etwas einfallen lassen. Und wenn es nicht klappt: etwas anderes probieren!



Technische Universität
Braunschweig

5-98

Schleifeninvariante (2/4)

Beispiel I (1/2)

|- $\{n > 0 \wedge q = 0 \wedge k = 1\}$
 while $k \neq n+1$ do $q := q+1; k := k+1$ od
 $\{q = n\}$

Problem: Finden der Schleifeninvariante

Schleifenvarianten gelten vor und nach *jeder* Ausführung des Schleifenrumpfes, wenn die Einstiegsbedingung gilt.

Im Beispiel:

- Schleife addiert pro Durchlauf 1 zum Wert der Variablen q und k hinzu
 - Startwert von q ist 0
 - Startwert von k ist 1, Abbruch bei $k=n+1$ (n Durchläufe)
 - Vor/nach jedem Durchlauf gilt somit $q=k-1$, zum Schluss $q=n$
- \Rightarrow Schleifeninvariante $\varphi \equiv q = k-1$
 \Rightarrow Kombiniert mit $\neg B = \neg(k \neq n+1) = (k = n+1)$ ergibt sich $\{q = n\}$

Iterationsregel:

$$\frac{\{\varphi \wedge B\} \alpha \{ \varphi \}}{\{\varphi\} \text{ while } B \text{ do } \alpha \text{ od } \{ \varphi \wedge \neg B \}}$$



Technische Universität
Braunschweig

5-99

Schleifeninvariante (3/4)

Beispiel I (2/2)

Und tatsächlich: $\varphi \equiv q = k-1$ bleibt vom Schleifenrumpf unbeeinträchtigt (bzw. ist invariant bzgl. des Schleifenrumpfes):

z.Z.: $\{q = k-1 \wedge k \neq n+1\} q := q+1; k := k+1 \{q = k-1\}$

Beweis:

Es gilt: $\{q = k-1 \wedge k \neq n+1\} \Rightarrow \{q+1 = k\} q := q+1 \{q = k\}$ // Zuweisung

$$\frac{\{q = k-1 \wedge k \neq n+1\} q := q+1 \{q = k\}}{\{q = k-1 \wedge k \neq n+1\} q := q+1 \{q = k\}}$$

// Anpassung

Ebenso: $\{q = k\} \Rightarrow \{q+1 = k+1\} k := k+1 \{q+1 = k\} \Rightarrow \{q = k-1\}$ // Zuweisung

$$\frac{\{q = k\} k := k+1 \{q = k-1\}}{\{q = k\} k := k+1 \{q = k-1\}}$$

// Anpassung

Zusammen (Sequenzregel):

$$\frac{\{q = k-1 \wedge k \neq n+1\} q := q+1 \{q = k\}, \{q = k\} k := k+1 \{q = k-1\}}{\{q = k-1 \wedge k \neq n+1\} q := q+1; k := k+1 \{q = k-1\}}$$



Technische Universität
Braunschweig

5-100

Schleifeninvariante (4/4)

Beispiel II

Suche x in einem Array $a[0..n-1]$, und zwar sequentiell von vorn nach hinten.

```
var array a[0..n-1] : int;
x, i : int; ... // irgendwas ...
i := 0; while i < n ∧ x ≠ a[i] do i := i + 1 od;
```

Iterationsregel:

$$\frac{\{ \varphi \wedge B \} \alpha \{ \varphi \}}{\{ \varphi \} \text{while } B \text{ do } \alpha \text{ od } \{ \varphi \wedge \neg B \}}$$

Finden einer geeigneten Schleifeninvariante:

- Schleife addiert *pro Durchlauf* 1 zum Wert der Variablen i hinzu
- Startwert von i ist 0
- Abbruch bei $i = n$ oder falls $x = a[i]$ (d.h. max. n Durchläufe; i (=Anzahl der Durchläufe) ist kleiner als n , falls x vor Position n gefunden wird)
- Vor/nach jedem Durchlauf gilt somit, dass x im Array a bis Position i (ausschließlich) noch nicht gefunden wurde, d.h.
 $\forall m : 0 \leq m < i \Rightarrow x \neq a[m]$ (\equiv Schleifeninvariante φ)
- zum Schluss gilt $(\forall m : 0 \leq m < i \Rightarrow x \neq a[m]) \wedge (i \geq n \vee x = a[i])$ gleichbedeutend mit $(\forall m : 0 \leq m < i \Rightarrow x \neq a[m]) \wedge (i < n \Rightarrow x = a[i])$



Technische Universität
Braunschweig

5-101

Beispiel (1/6)

Der folgende Algorithmus berechnet die ganzzahlige Wurzel bei Eingabe x größer oder gleich 0 (s. Kapitel 4.2).

```
XYZ: var w,x,y,z : int;
input x;
z := 0; w := 1; y := 1;
while w ≤ x od
  z := z + 1; w := w + y + 2; y := y + 2 od
output z
```

Aber tut er das tatsächlich? Wir wollen es beweisen! Zu zeigen:

```
{ x ≥ 0 } z := 0; w := 1; y := 1;
while w ≤ x od
  z := z + 1; w := w + y + 2; y := y + 2 od
{ z² ≤ x < (z+1)² }
```

// gleichbedeutend mit , aber einfacher
// zu beweisen



Technische Universität
Braunschweig

5-102

Beispiel (2/6)

Beweis (Skizze): Seien

 $\beta \equiv z := 0; w := 1; y := 1$
 $B \equiv w \leq x$
 $\alpha \equiv z := z + 1; w := w + y + 2; y := y + 2$
 $\psi \equiv \{ z^2 \leq x < (z+1)^2 \}$

Dann ist zu zeigen: $\{ x \geq 0 \} \beta; \text{while } B \text{ do } \alpha \text{ od } \{ \psi \}$

$$\frac{\{ x \geq 0 \} z := 0; w := 1; y := 1; \text{while } w \leq x \text{ do } z := z + 1; w := w + y + 2; y := y + 2 \text{ od } \{ z^2 \leq x < (z+1)^2 \}}{\{ x \geq 0 \} z := 0; w := 1; y := 1; \text{while } w \leq x \text{ do } z := z + 1; w := w + y + 2; y := y + 2 \text{ od } \{ z^2 \leq x < (z+1)^2 \}}$$

Das Problem ist die Schleife: um die Iterationsregel anzuwenden, müssen wir eine passende **Schleifeninvariante** finden (s.o.), d.h. eine Formel φ , für die gilt:

damit könnten ...

- $\{ x \geq 0 \} \beta \{ \varphi \}$
 - $\{ \varphi \wedge B \} \alpha \{ \varphi \}$
 - $(\varphi \wedge \neg B) \Rightarrow \psi$
1. Iteration: $\{ \varphi \} \text{while } B \text{ do } \alpha \text{ od } \{ \varphi \wedge \neg B \}$
 2. Sequenz: $\{ x \geq 0 \} \beta; \text{while } B \text{ do } \alpha \text{ od } \{ \varphi \wedge \neg B \}$
 3. Anpassung: $\{ x \geq 0 \} \beta; \text{while } B \text{ do } \alpha \text{ od } \{ \psi \}$ (fertig)

... wir schließen



Technische Universität
Braunschweig

5-103

Beispiel (3/6)

Schleifenvarianten gelten vor *und* nach jeder Ausführung des Schleifenumpfes, wenn die Einstiegsbedingung gilt.

Am Ende der Iteration spiegelt die Schleifenvariante zudem das Endergebnis der Berechnung wider.

$$\frac{\{ x \geq 0 \} z := 0; w := 1; y := 1; \text{while } w \leq x \text{ do } z := z + 1; w := w + y + 2; y := y + 2 \text{ od } \{ z^2 \leq x < (z+1)^2 \}}{\{ x \geq 0 \} z := 0; w := 1; y := 1; \text{while } w \leq x \text{ do } z := z + 1; w := w + y + 2; y := y + 2 \text{ od } \{ z^2 \leq x < (z+1)^2 \}}$$

Eine geeignete (und nicht wirklich triviale) Schleifeninvariante ist

$$\varphi \equiv z^2 \leq x \wedge (z+1)^2 = w \wedge 2z+1 = y.$$

Nach Ausführung der Schleife gilt $w = (z+1)^2 > x$ und durch Anpassung dann auch das Gewünschte: $z^2 \leq x < (z+1)^2$.

Ist die Schleifeninvariante erst einmal gefunden, ist es damit leicht, die partielle Korrektheit zu beweisen:

$$\models_P \{ x \geq 0 \} XYZ \{ z^2 \leq x < (z+1)^2 \}$$



Technische Universität
Braunschweig

5-104

Beispiel (4/6)

Zur übersichtlichen Notation verwenden wir ein *Tableau*.

Die Regel für Sequenz und der vorherrschende Aufbau von Programmen aus (Teil-)Sequenzen geben Anlass zu dieser ökonomischeren Notation für einen Beweis der partiellen Korrektheit (auch *Beweis-Tableau*):

Regel Tableau-Schreibweise

$$\frac{\{ \varphi_1 \} \alpha_1 \quad \{ \varphi_2 \}, \{ \varphi_2 \} \alpha_2 \quad \{ \varphi_3 \}}{\{ \varphi_1 \} \alpha_1; \alpha_2 \quad \{ \varphi_3 \}} \quad \text{Sequenz} \rightarrow \begin{matrix} \{ \varphi_1 \} \\ \alpha_1; \\ \alpha_2 \\ \{ \varphi_3 \} \end{matrix}$$

$$\frac{\{ \varphi_1 \} \quad \{ \varphi_2 \}}{\{ \varphi_1 \} \Rightarrow \varphi_2} \quad \text{Anpassung} \rightarrow \begin{matrix} \{ \varphi_1 \} \\ \{ \varphi_2 \} \end{matrix}$$

Beispiel (5/6)

Beispiel XYZ:

```

{ x ≥ 0 }
  { x ≥ 0 ∧ 0 = 0 }
  z := 0;
  y := 1;
  w := 1;
  while w ≤ x do
    { x ≥ 0 ∧ z = 0 ∧ y = 1 ∧ w = 1 }
    { x ≥ 0 ∧ z = 0 ∧ y = 1 ∧ 1 = 1 }
    { x ≥ 0 ∧ z = 0 ∧ y = 1 ∧ w = 1 }
    { z ≤ x ∧ (z+1)2 = w ∧ 2z+1 = y } // Schleifeninvariante φ !!
    while w ≤ x do
      { w ≤ x ∧ z ≤ x ∧ (z+1)2 = w ∧ 2z+1 = y }
      { w ≤ x ∧ (z+1)2 ≤ x ∧ (z+1)2 = w ∧ 2(z+1)+1 = y + 2 }
      z := z + 1;
      { w ≤ x ∧ z ≤ x ∧ z2 = w ∧ 2z+1 = y + 2 }
    
```

```

{ x ≥ 0 } z := 0; w := 1; y := 1;
  while w ≤ x do
    z := z + 1;
    w := w + y + 2;
    y := y + 2 od
  { z2 ≤ x < (z+1)2 }

```

```

Iterationsregel:
  { φ ∧ B } α { φ }
  { φ } while B do α od { φ ∧ ¬ B }

```

Beispiel (6/6)

Beispiel XYZ:

```

...
z := z + 1;
{ w ≤ x ∧ z2 ≤ x ∧ z2 = w ∧ 2z + 1 = y + 2 }
{ w + y + 2 ≤ x + y + 2 ∧ z2 ≤ x ∧ (z+1)2 = w + y + 2 ∧ 2z + 1 = y + 2 }
// (z+1)2 = z2 + 2z + 1 = z2 + y + 2 = w + y + 2
w := w + y + 2;
{ w ≤ x + y + 2 ∧ z2 ≤ x ∧ (z+1)2 = w ∧ 2z + 1 = y + 2 }
y := y + 2;
{ w ≤ x + y + 2 ∧ z2 ≤ x ∧ (z+1)2 = w ∧ 2z + 1 = y }
{ z2 ≤ x ∧ (z+1)2 = w ∧ 2z + 1 = y } // et voilà: Schleifeninvariante φ
od // Sie erinnern sich: das Ende der While-Schleife ;-)
{ z2 ≤ x ∧ (z+1)2 = w ∧ 2z + 1 = y ∧ ¬ w ≤ x }
{ z2 ≤ x < (z+1)2 = w ∧ 2z + 1 = y ∧ w > x }
{ z2 ≤ x < (z+1)2 } (fertig!)

```

Konsistenz und Vollständigkeit

Für jede Logik mit einem Begriff von Gültigkeit \models und einem System von Beweisregeln (*Kalkül*) \vdash sind folgende Fragen von Interesse:

Konsistenz: ist \vdash konsistent bzgl. \models ?

d.h. folgt aus $\varphi \vdash \neg \varphi$ immer $\varphi \models \varphi$?

"ist alles, was sich herleiten lässt, auch wahr?"

Vollständigkeit: ist \vdash vollständig bzgl. \models ?

d.h. folgt aus $\varphi \models \varphi$ immer $\varphi \vdash \varphi$?

"lässt sich alles, was wahr ist, auch herleiten?"

In einem realistischen System (nicht unsere Spielzeugsprache) sind die Hoare'schen Beweisregeln stets konsistent, aber niemals vollständig (> K.Gödel 1934 (Unvollständigkeitssatz)). Sie sind jedoch meist relativ vollständig, d.h. vollständig bzgl. eines eingeschränkten Gültigkeitsbegriffs (Cook 1978).

Wir können diesen Fragen hier nicht nachgehen -zumal wir auch die Gültigkeit \models nicht formal präzisiert haben. Dies geschieht im Rahmen der denotationalen Semantik von Programmiersprachen.

Totale Korrektheit (1/2)

Wie lässt sich totale Korrektheit $\models_{\tau} \{ \varphi \} \alpha \{ \psi \}$ beweisen?

Hierzu muss zunächst partielle Korrektheit $\models_p \{ \varphi \} \alpha \{ \psi \}$ bewiesen werden, und darüber hinaus (s. o.):

$$\varphi \Rightarrow \alpha \text{ terminiert}$$

Grundannahmen:

- die Auswertung eines Ausdrucks (Terms) terminiert immer
- eine Wertzuweisung $x := w$ terminiert immer
- Sequenz: α_1, α_2 terminiert genau dann, wenn α_1 und α_2 terminieren
- Verzweigung: **if B then** α_1 **else** α_2 **fi** terminiert genau dann, wenn die gewählte Alternative α_1 bzw. α_2 terminiert



Technische Universität
Braunschweig

5-109

Totale Korrektheit (2/2)

Problemfall Iteration while B do α od :

Auch wenn α terminiert ist dadurch nicht automatisch gegeben, dass die Schleifenkonstruktion terminiert.

Beispiel: while $x \neq 0$ **do** $x := x-1$ **od** terminiert nicht bei Startwerten $x < 0$, auch wenn $x := x-1$ immer terminiert.

Daraus folgt die naheliegende **Beweismethode für totale Korrektheit:**

1. beweise partielle Korrektheit
2. beweise für jede Schleife, dass sie nur endlich oft durchlaufen werden kann d.h. terminiert.

Aber wie beweist man letzteres?



5-110



Technische Universität
Braunschweig

Terminierung von Schleifen (1/2)

Um Terminierung von Schleifen zu beweisen, gibt es kein allgemeines Rezept. Oft führt jedoch folgende Methode zum Ziel:

1. *suche* Ausdruck u , dessen Wert eine natürliche Zahl ≥ 0 ist
 2. *beweise:* der Wert von u wird bei jedem Schleifendurchlauf kleiner.
- Aus letzterem folgt, dass es nur endlich viele Schleifendurchläufe geben kann, die Schleife also terminieren muss.

Beispiel: XYZ (s.o.)

{ $x \geq 0$ } $z := 0; w := 1; y := 1;$

while $w \leq x$ **do**

$z := z + 1; w := w + y + 2; y := y + 2$ **od**

Behauptung:

Die while -Schleife terminiert, sofern die Vorbedingung erfüllt ist (Tatsächlich terminiert sie immer, aber das brauchen wir für die totale Korrektheit nicht zu beweisen).



Technische Universität
Braunschweig

5-111

Terminierung von Schleifen (2/2)

Beweis:

Sei $u = x - w$. Es gilt

1. $w \leq x \Rightarrow u \geq 0$,
d.h. u bleibt bei allen
Schleifendurchläufen nicht negativ.

2. u wird bei jedem Schleifendurchlauf kleiner:

sei u der Wert zu Beginn eines Schleifendurchlaufs, u' der danach. Dann gilt:

$$u = x - w$$

$$u' = x - (w+y+2) = x - w - y - 2$$

Da stets $y > 0$ ist, ist $u' < u$

Z.z.: folgende while-Schleife terminiert für $\{ x \geq 0 \}$:

$z := 0; w := 1; y := 1;$
while $w \leq x$ **do**

$z := z + 1; w := w + y + 2; y := y + 2$ **od**

Methode:

1. *suche* Ausdruck u , dessen Wert eine natürliche Zahl 0 ist
2. *beweise:* der Wert von u wird bei jedem Schleifendurchlauf kleiner.



Technische Universität
Braunschweig

5-112