

Einführung in die Informatik 1

– Algorithmen und algorithmische Sprachkonzepte –

Sven Kosub

AG Algorithmik/Theorie komplexer Systeme
Universität Konstanz

E 202 | Sven.Kosub@uni-konstanz.de | Sprechstunde: Dienstag, 16:30-18:00 Uhr, o.n.V.

Wintersemester 2009/2010

Informatik =

Wissenschaft von der systematischen Verarbeitung von Informationen,
insbesondere ...

... der automatischen Verarbeitung mit Hilfe von Rechenanlagen
= Computerwissenschaft

zentrale Gegenstände der Informatik:

- Information
- Algorithmen (\simeq Systematik der Verarbeitung)
- Computer (\simeq Rechenanlagen)

✓
heute
✓

Algorithmus:

- präzise festgelegtes Verfahren zur Lösung von Problemen (bzw. einer Klasse von Problemen), das aus endlich vielen, effektiv ausführbaren elementaren Lösungsschritten besteht
- Problemklasse ist Menge von Eingabe-Ausgabe-Paaren

Algorithmen müssen folgenden Bedingungen genügen:

- Spezifikation
- Durchführbarkeit
- Korrektheit

- **Eingabespezifikation:**

Es muss genau spezifiziert sein, welche Eingabegrößen erforderlich sind und welchen Anforderungen diese Größen genügen müssen, damit das Verfahren funktioniert

- **Ausgabespezifikation:**

Es muss genau spezifiziert sein, welche Ausgabegrößen (Resultate) mit welchen Eigenschaften berechnet werden

- **endliche Beschreibung:**

Verfahren muss in einem endlichen Text vollständig beschrieben sein

- **Effektivität:**

Jeder Schritt des Verfahrens muss effektiv (d.h. tatsächlich) „mechanisch“ ausführbar sein

→ Beachte: Effektivität \neq Effizienz

- **Determiniertheit:**

Verfahrensablauf ist zu jedem Zeitpunkt fest vorgeschrieben

- **partielle Korrektheit:**

jedes berechnete Ergebnis genügt der Ausgabespezifikation, sofern die Eingaben der Eingabespezifikation genügen

- **Terminierung:**

Algorithmus hält nach endlich vielen Schritten mit einem Ergebnis an, sofern die Eingabe der Eingabespezifikation genügt

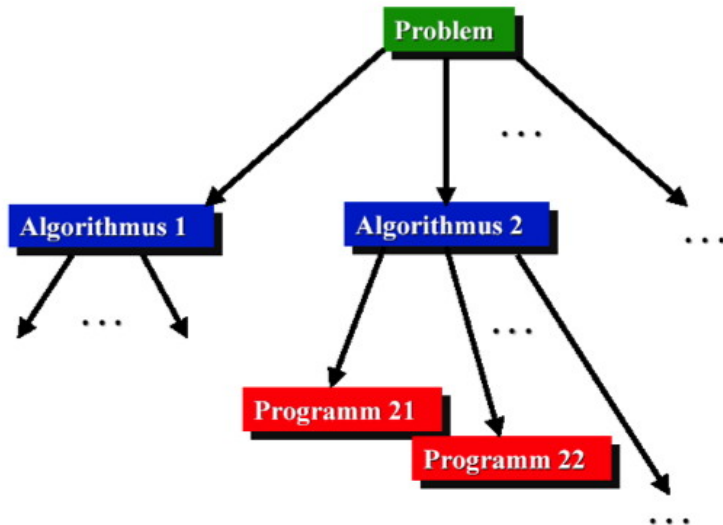
- Computer führen nur in einer formalen Sprache beschriebene Algorithmen aus
- **Programmiersprache** = formale Sprache für einen Computer
- **Programm** = Beschreibung des Algorithmus in Programmiersprache

- Programm repräsentiert nur einen Algorithmus
- Algorithmus wird durch viele verschiedene (i.A. unendlich viele) Programme repräsentiert

- Programmierung setzt Algorithmenentwurf voraus:

Kein Programm ohne Algorithmus!

Problem-Algorithmus-Programm



Form der Algorithmenbeschreibung:

- Alltagssprache
- konkrete Programmiersprache
- Pseudocode

Elemente der Algorithmenbeschreibung:

- Folgen einzelner Bearbeitungsschritte
- Schrittwiederholung durch **Iteration**: Rückverweise in der Folge wie z.B. „Weiter mit Schritt (2)“
- Schrittwiederholung durch **Rekursion**: Wiederaufruf des Algorithmus mit einfacherer Problemstellung
- Sprünge (bedingt/unbedingt)

Grundschema des Algorithmenaufbaus:

Name des Algorithmus und Liste von Parametern

Spezifikation der Eingaben

Spezifikation der Ausgaben

- 1 **Vorbereitung:** Einführung von Hilfsgrößen etc.
- 2 **Trivialfallbehandlung:** Prüfe, ob ein einfacher Fall vorliegt; falls ja, dann Beendigung mit Ergebnis
- 3 **Arbeit (Problemreduktion, Ergebnisaufbau):** Reduziere Problemstellung X auf eine einfachere Form X' mit $X > X'$ bzgl. einer (wohlfundierten) Ordnung $>$ und baue entsprechend der Reduktion einen Teil des Ergebnis auf
- 4 **Rekursion** bzw. **Iteration:** Rufe zur Weiterverarbeitung den Algorithmus mit dem reduzierten X' erneut auf (Rekursion) bzw. fahre mit X' statt X bei Schritt 2 fort (Iteration)

Form der Algorithmenbeschreibung (verfeinert):

- elementar-iterativ
- Flussdiagramme
- strukturiert-iterativ
- rekursiv
- Pseudocode

Beschreibe ein Verfahren zur Berechnung des Divisionrestes r bei Division von a durch b , d.h. finde $0 \leq r < b$ mit $a = t \cdot b + r$ für geeignetes t

elementar-iterative Beschreibungsform:

- Iterationen werden durch Sprünge realisiert

Algorithmus zur Berechnung von $\text{mod}(a, b)$:

$\text{mod}(a, b)$

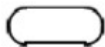
// Anforderungen (Eingabe): $a, b \in \mathbb{Z}, a \geq 0, b > 0$

// Zusicherung (Ausgabe): Resultat ist Divisionsrest von $\frac{a}{b}$

- 1 Kopiere a nach r
- 2 Prüfe, ob $r < b$ gilt; falls ja, dann gib Resultat r aus
- 3 Subtrahiere b von r und speichere Resultat wieder in r
- 4 Weiter mit Schritt 2

Flussdiagramme:

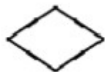
- graphische Darstellung des Steuerungsverlaufs
- Komponenten von Flussdiagrammen:



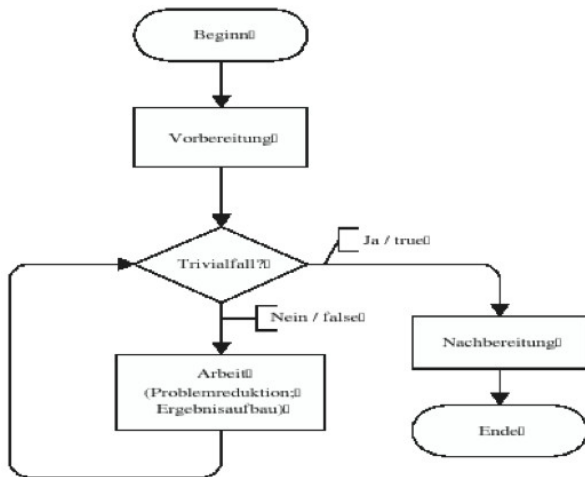
Beginn / Ende des Algorithmus



Anweisungen, Operationen



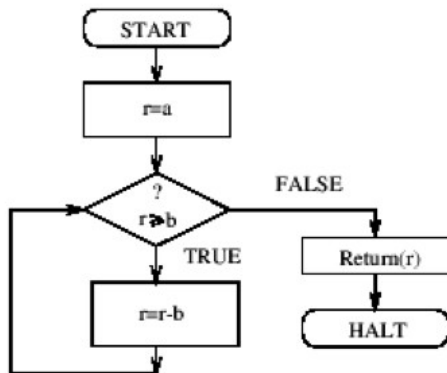
Verzweigungen: Der weitere Verlauf hängt vom Wahrheitswert des Ausdrucks im Inneren ab.



algorithmisches Grundschema als Flussdiagramm

Algorithmen: Beschreibungsformen

Flussdiagramm für den Algorithmus $\text{mod}(a, b)$

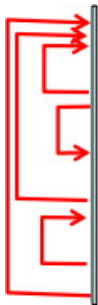


Kontrollfluss für $\text{mod}(7,3)$

- START
- $a = 7, b = 3$
- $r = 7$
- $7 \geq 3?$ TRUE
- $r = 7 - 3$
- $4 \geq 3?$ TRUE
- $r = 4 - 3$
- $1 \geq 3?$ FALSE
- RETURN(1)
- HALT

strukturiert-iterative Beschreibungsform:

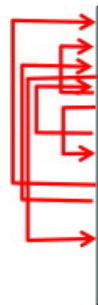
- Sprünge dürfen nur ineinander geschachtelt sein, d.h. Schleifen überkreuzen sich nicht



strukturiert-iterativ



elementar-iterativ



Spaghetti-Code

Sprungbehandlung bei strukturiert-iterativer Beschreibungsform:

- keine unbedingten Sprünge (kein `goto ...`)
- Sprünge werden nur implizit angegeben
- Sprünge Teil höhere Iterationsstrukturen
 - Fallunterscheidung (`if ... then ... else ...`)
 - Schleifen (`while ... do ...`)

Java erlaubt strukturiert-iterative Beschreibung (kein `goto` in Java!)

Fallunterscheidung in Java:

- **syntaktisch:**

```
if (Bedingung) { Anweisungssequenz 1 }  
else { Anweisungssequenz 2 }
```

- **semantisch:**

Ist Bedingung erfüllt (`true`), dann führe Anweisungssequenz 1 aus,
sonst führe Anweisungssequenz 2 aus

Schleifen in Java:

- **syntaktisch:**

```
while (Bedingung) { Anweisungssequenz }
```

- **semantisch:**

Bei Eintritt in die `while`-Schleife wird Bedingung überprüft

Ist Bedingung erfüllt (`true`), führe die Anweisungssequenz aus und gehe zum Test der Bedingung zurück

Ist Bedingung nicht erfüllt (`false`), verlasse die `while`-Schleife ohne Ausführung der Anweisungssequenz

Beachte: `while`-Schleife entspricht der Konstruktion (falls `goto` vorhanden)

```
M: if (Bedingung ) { Anweisungssequenz; goto M; }
```

Algorithmen: Beschreibungsformen

strukturiert-iterativer Algorithmus für die mod-Funktion (in Java):

```
int mod(int a, int b) {
    /* Anforderungen (Eingabe): a: a>=0; b: b>0 */
    /* Zusicherung (Ausgabe): r: a=t*b + r und 0<r<=b */
    /* 1. Vereinbarungen */
    int r;
    /* 2. Initialisierungen */
    r=a;
    /* 3. Iterationsbeginn */
    while (r>=b) {
        /* 4. Problemreduktion */
        r=r-b;
    }
    /* 5. Iterationsende */
    /* 6. Ergebnisrückgabe */
    return(r);
}
```

rekursive Beschreibungsform:

1 Basis:

Gib eine direkte Lösung für den Fall an, dass Eingabe X einfach ist

2 Rekursionsschritt:

Führe Lösung für Problem $P(X)$ für eine komplexe Eingabe X durch einen Schritt der Problemreduktion auf die Lösung des gleichen Problems $P(X')$ für eine einfachere Eingabe X' zurück

Dabei muss $X > X'$ für eine wohlfundierte Ordnung $>$ gelten

rekursive Definition der mod-Funktion (für natürliche Zahlen a und $b > 0$):

$$\text{mod}(a, b) =_{\text{def}} \begin{cases} a & \text{falls } a < b \\ \text{mod}(a - b, b) & \text{falls } a \geq b \end{cases}$$

rekursiver Algorithmus für die mod-Funktion (in Java):

```
int mod(int a, int b) {  
    /* Anforderungen (Eingabe): a: a>=0; b: b>0 */  
    /* Zusicherung (Ausgabe): r: a=t*b + r und 0<r<=b */  
    /* 1. Vereinbarungen */  
    int r;  
    /* 2. Initialisierungen */  
    r=a;  
    /* 3. Einfacher Fall */  
    if (r<b) { return(r); }  
    /* 4. Problemreduktion */  
    r=r-b;  
    /* 5. Rekursionsschritt */  
    return(mod(r,b));  
}
```

Pseudocode:

- Mischform aus Syntax höherer Programmiersprachen und natürlicher Sprache
- typische Elemente von Programmiersprachen: `if ...`, `while ...`
- Trade-off zwischen Anschaulichkeit (natürliche Sprache) und Formalisierung (Programmiersprache)

```
if (keine Elemente mehr zu sortieren) { return(Fertig) }
```

Euklidischer Algorithmus:

- Berechnung des ggT zweier natürlicher Zahlen, z.B.

$$\text{ggT}(36, 120) = 12, \quad \text{da } 36 = 2^2 \cdot 3^2 \text{ und } 120 = 2^3 \cdot 3 \cdot 5$$

- rekursive (mathematische) Definition

$$\text{ggT}(a, b) =_{\text{def}} \begin{cases} a & \text{falls } b = 0 \\ \text{ggT}(b, a) & \text{falls } b > a \\ \text{ggT}(b, \text{mod}(a, b)) & \text{sonst} \end{cases}$$

ergibt für $\text{ggT}(36, 120)$:

$$\begin{aligned} \text{ggT}(36, 120) &= \text{ggT}(120, 36) && \text{ („} b > a \text{“)} \\ &= \text{ggT}(36, \text{mod}(120, 36)) = \text{ggT}(36, 12) && \text{ („sonst“)} \\ &= \text{ggT}(12, \text{mod}(36, 12)) = \text{ggT}(12, 0) && \text{ („sonst“)} \\ &= 12 && \text{ („} b = 0 \text{“)} \end{aligned}$$

Euklidischer Algorithmus (Forts.):

- rekursiver Algorithmus in Pseudo-Code:

Algorithmus: $\text{ggT}(a, b)$

Eingabe: natürliche Zahlen a und b

Ausgabe: größter gemeinsamer Teiler von a und b

(1) if ($b==0$) { return (a) };

/* Trivialfall

(2) if ($b>a$) { return ($\text{ggT}(b, a)$);

/* einfacher Fall

(3) return ($\text{ggT}(b, \text{mod}(a, b))$);

/* Reduktion und Rekursion

- Warum terminiert der Euklidische Algorithmus?

- Eingaben sind Paare (a, b) mit $a, b \in \mathbb{N}$
- $(a, b) > (a', b') \iff_{\text{def}} a + b > a' + b'$ oder $a + b = a' + b' \wedge a < a'$
- Rekursionsaufrufe bei (a, b) erfolgen mit (a', b') , sodass $(a, b) > (a', b')$

$$(36, 120) > (120, 36) > (36, 12) > (12, 0)$$

- für jedes (a, b) gibt es nur endlich viele (a', b') mit $(a', b') < (a, b)$