Algorithmen und Datenstrukturen
(Informatik II)

Prof. Dr. Hans-Jürgen Buhl

2001

Fachbereich Mathematik (7)
Institut für Angewandte Informatik
Bergische Universität – Gesamthochschule Wuppertal

Interner Bericht der Integrierten Arbeitsgruppe
Mathematische Probleme aus dem Ingenieurbereich
IAGMPI – 9702
November 1999

Praktische Informatik 04

Inhaltsverzeichnis

1 Algorithmen **1**
1.1 Ein intuitiver Algorithmenbegriff 1
1.2 Fallstudien zur Computersystem- und Softwarequalität (oder-misere?) 6
1.3 Qualitätsanforderungen an Software 8
1.3.1 Produktorientierte Gütekriterien 8
1.3.2 Projektorientierte Gütekriterien 8
1.3.3 Spezifikation und Verifikation von Code 9
1.3.4 Explizite und implizite Spezifikation von Funktionen . 12
1.3.5 Spezifikation von Bausteinen von Softwaresystemen . . 13

Abbildungsverzeichnis

1.1 Flussdiagramm zum euklidischen Algorithmus 3
1.2 Struktogramm zum euklidischen Algorithmus 4

Tabellenverzeichnis

1.1 Zahlenbeispiel zum euklidischen Algorithmus 2

Kapitel 1

Algorithmen

1.1 Ein intuitiver Algorithmenbegriff

Definition 1.1.1
Ein *Algorithmus* ist eine endliche Folge von eindeutigen Anweisungen, mittels derer in endlich vielen Schritten aus vorgegebenen spezifizierten Eingabegrößen spezifizierte Ergebnisgrößen gewonnen werden.
Ein Algorithmus berechnet also eine Funktion der Eingabe.
Dabei müssen folgende Punkte erfüllt sein:

1. Ein- und Ausgabe sind genau festgelegt; zu jeder Eingabe gibt es (genau) eine gültige Ausgabe.

   Frage : „Welche Eingabegrößen sind erlaubt und/oder sinnvoll ?“

   Frage : „Welche Funktion soll der Algorithmus berechnen ?“
   (→ Problemspezifikation)

   Frage : „Berechnet der Algorithmus wirklich die spezifizierte Funktion ?“
   (→ Korrektheitsuntersuchung)

2. Jede Anweisung darf nur endlich viele Schritte benötigen und nur endlich oft ausgeführt werden.

   Frage : „Wie sehen die Schritte aus ?“
   (→ Maschinenmodell)

   Frage : „Terminiert der Algorithmus, d.h. liefert er nach endlich vielen Schritten eine Ausgabe ?“
   (→ Terminierung)

3. Jede Anweisung muß ein eindeutiges (reproduzierbares) Resultat haben.
   (→ Definitheit)

4. Das Aufschreiben des Algorithmus darf nur endlich viel Platz beanspruchen (also nicht: „usw.“).

5. Möglichst geringer Ressourcenverbrauch wie Speicher, Rechenzeit.
   (→ Effizienz)

6. Der Algorithmus beruht auf einer nachvollziehbaren Idee und ist verständlich formuliert.
   (→ Verständlichkeit)

Beispiel: (Euklidischer Algorithmus)
Geg.: Zwei Zahlen $m, n \in \mathbb{N}$, $m > n$
Ges.: Der größte gemeinsame Teiler $ggT(m, n)$

1. [Division mit Rest]
   Berechne $m = n \cdot q + r$, $r, q \in \mathbb{N}_0$, $0 \leq r < n$

2. [Ergebnis]
   Falls $r = 0$ beende Algorithmus, $ggT = n$

3. [Ersetzen]
   $m \leftarrow n$, $n \leftarrow r$, gehe zu 1.

(”←” heißt: „wird ersetzt durch“)

Zahlenbeispiel: $m = 2754$, $n = 378$

Tabelle 1.1: Zahlenbeispiel

Status nach	Schritt-Nr.	m	n	q	r	ggT
Startwerte	-	2754	378	?	?	?
1. Durchlauf	1.	2754	378	7	108	?
	3.	378	108	7	108	?
2. Durchlauf	1.	378	108	3	54	?
	3.	108	54	3	54	?
3. Durchlauf	1.	108	54	2	0	?
	2.	108	54	2	0	54

Hier nun das zugehörige *Flussdiagramm*:

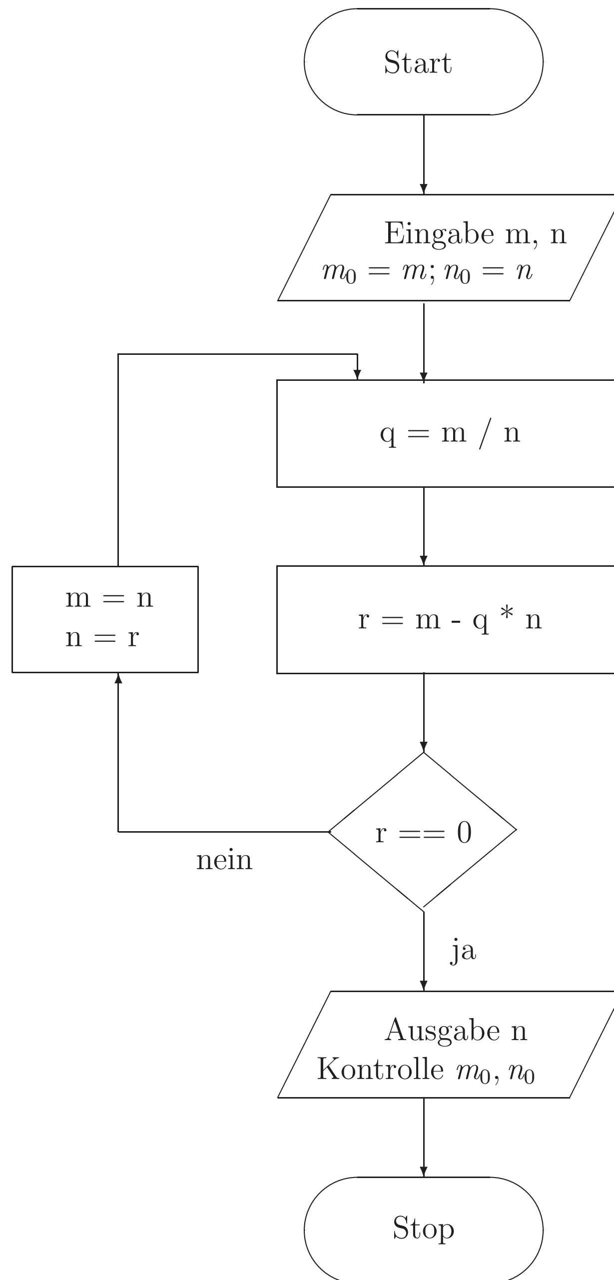
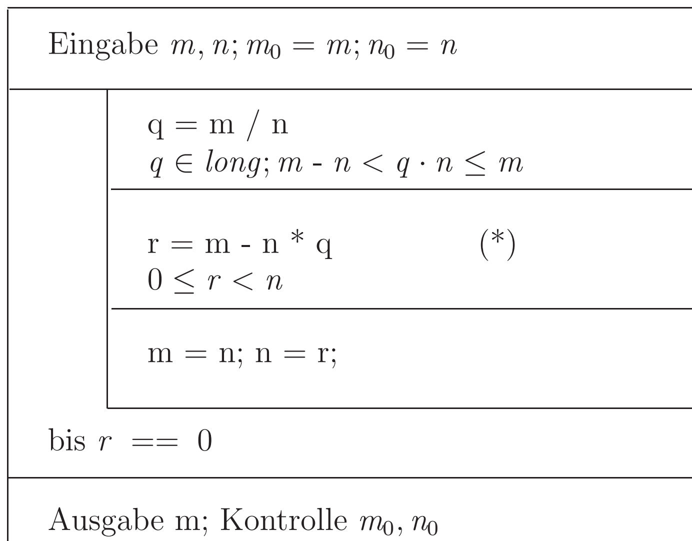


Abbildung 1.1: Flussdiagramm

oder besser noch ein Struktogramm:

Abbildung 1.2: Struktogramm



- *Spezifikation* hier:
  Eingabe : $m, n \in \mathbb{N},\ m > n$
  Ausgabe : $ggT = ggT(m, n) \in \mathbb{N}$, d.h. diejenige natürliche Zahl, die n und m teilt und für die gilt : Teilt $d \in \mathbb{N}$ sowohl n als auch m, so ist d Teiler von ggT.

- *Korrektheit:* Ist ggT aus dem Algorithmus wirklich der ggT aus der Spezifikation? Zu zeigen:

  – teilt d die Eingabegrößen n und m, so teilt d auch jeweils die n, m, die in Schritt 3 erzeugt werden.
  ($\rightarrow$ d teilt ggT)

  – ggT aus Schritt 2 ist gemeinsamer Teiler aller n, m, die im Algorithmus vorkommen.

  Oder einfacher die Invariante $ggT(m, n) = ggT(n, r)$ in (*) von Abb. 1.2.

- *Terminierung:* Zu zeigen: $r = 0$ wird nach endlich vielen Durchläufen erreicht. Ist r_i der Wert von r in Schritt 2 im i-ten Durchlauf, so gilt wegen Schritt 1 und 3

$$r_i < r_{i\text{-}1}$$

mit $r_0 = n$. Wegen $r_i \in \mathbb{N}_0$ folgt so: Der Algorithmus terminiert nach spätestens n Durchläufen. Eine solche strikt monoton fallende Größe nennt man *(Schleifen-)Variante*.

- *Definierte Schritte:* Einzige „Schwierigkeit“: Division mit Rest. Hierzu existiert ein „Elementaralgorithmus“ (siehe C-Standard).

- *Definitheit:* Zu zeigen: $n, m \in \mathbb{N}$ gilt stets in Schritt 1 (ansonsten ist Division mit Rest nicht definiert).
  Beweis: Richtig für Eingabe („nach 0. Durchlauf“): Gilt $n, m \in \mathbb{N}$ in Schritt 3 im i-ten Durchlauf, so gilt im $(i+1)$-ten Durchlauf:

  $$m = n \cdot q + r, \quad 0 \leq r < n, \quad r \in \mathbb{N}_0$$

  Ist $r = 0$, so terminiert der Algorithmus in Schritt 2, ansonsten ist $r \in \mathbb{N}$, $n \in \mathbb{N}$ und damit in Schritt 3 im $(i+1)$-ten Durchlauf auch $m, n \in \mathbb{N}$.

- *Effizienz:* Speicher : 4 natürliche Zahlen, Rechenzeit : „nicht schlecht“ (besser als bei Ausnutzung der Invarianten $ggT(m,n) = ggT(n,r)$.

1.2 Fallstudien zur Computersystem- und Softwarequalität (oder -misere?)

Fehlerhafte Spezifikation/Korrektheit/Definitheit bei:

- Untergang der „Sleipner A“ Ölplattform
- Verlust des „Mars Climate Orbiter“
- Fluggesellschaftsroutenbuchungen auf „gut Glück“
- PLZ in Wuppertal (fehlerhafte Spezifikation)
- neues Sommerzeitende und Terminkalender
- $\{0, \ldots, 99\}$ als Jahreszahlen
- Ampelsteuerung
- Fehlalarm im Kanaltunnel
- Berliner Magnetbahn (fehlende Plausibilitätsüberlegungen)
- Ausfall der „Telefon“-Computer
- Glücksspiel und „Zufallszahlen“
- AOL offline
- DB bucht doppelt
- falsche Telefontarife (Image-Verlust, Klage gegen SW-Lieferanten)
- Postbank: falsche Zinsberechnung
- Flughafen Düsseldorf-Luftfrachtzentrum: $\geq$ 70.000 DM für Aushilfen
- Einschaltquote GfK
- Geschlossene Türen in Glasgow (mechanische Notsysteme? Konsistenzüberlegungen?)
- THERAC-25
- Flugzeug-Schleudersitz

- A 320 in Warschau: Bremssystem zu intelligent?
- A 300 in China : Copilot gegen SW ($\rightarrow$ 264 Tote)
- Ungenügende Unsicherheitsinfos von errechneten Ergebnissen: AEGIS
- Rundungsfehler und die Raketenabwehr im Irak-Krieg
- Wann ist $1.407\ldots = 0.64$?
- Pentium FPU-Fehler (HW-Spezifikationsfehler oder fehlende Tests?)
- Ariane 5 Explosion

1.3 Qualitätsanforderungen an Software

1.3.1 Produktorientierte Gütekriterien

1. Funktionale Korrektheit
2. Funktionale Vollständigkeit
3. Robustheit gegenüber dem Benutzer
4. Benutzerfreundlichkeit
5. Effizienz in Laufzeit
6. Effizienz im Arbeitsspeicherbedarf
7. Effizienz im Plattenplatzbedarf
8. Integrität (gegenüber unautorisierten Änderungen)
9. Kompatibilität/Integrationsfähigkeit/Erfüllen von Standards

1.3.2 Projektorientierte Gütekriterien

1. Überprüfbarkeit
2. Verständlichkeit
3. Wartbarkeit
4. Änder- und Erweiterbarkeit
5. Portierbarkeit
6. Wiederverwendbarkeit insbesondere von Teilproblemlösungen

1.3.3 Spezifikation und Verifikation von Code

Kommentieren sie den folgenden Quellcode

```
//////////////////////////////////
// Datei:   power.cc
// Version: 1.0
// Zweck:   while-Schleife
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
//////////////////////////////////

#include        <iostream>
#include        <iomanip>

using namespace std;

double power2(double x, int exp)
{
  double erg(1.0);

  if (exp < 0)
        throw "negativer Exponent bei power2 nicht erlaubt!";
  while ( exp > 0 ) {
        if ((exp % 2 ) != 0) {
                erg *= x;
                exp--;
        } else {                // hier ist exp gerade
                x = x*x;
                exp = exp/2;
        }
  };
  return erg;
};

int main()
{
 cout << setprecision(10) << power2(13.5, 3) << endl;

 return 0;
}
```

durch Angabe von Schleifeninvariante, -variante, Vor- und Nachbedingung, ... nach dem folgenden Muster:

```
//////////////////////////////////
// Datei:   power.cc
// Version: 0.91
// Zweck:   while-Schleife
// Autor:   Hans-Juergen Buhl
// Datum:   15.09.1998
//////////////////////////////////

#include <iostream>
#include <iomanip>

using namespace std;

double power2(double x, int exp)
//                                            (Spezifikation)
//   power2:  double x int ---> double
//      exp < 0:   Exception "negativer Exponent bei power2 nicht erlaubt!"
//      exp > 0:   power2(x, exp) == (x ^ exp) * (1 + eps), abs(eps) klein
//
{
  // Sei x0 = x, exp0 = exp

  double erg(1.0);

  // erg == 1.0
  // x0^exp0 == erg * x^exp

  if (exp < 0)
        throw "negativer Exponent bei power2 nicht erlaubt!";

  // exp >= 0

  for (int i = exp; i > 0; i--){
        //
        // i in int, i <= exp, i > 0
        //
        //     x0^exp0 == erg * x^i
        //
```

```
        erg *= x;

        // Schleifeninvariante:
        //
        //     x0^exp0 == erg * x^(i-1), i-1 >= 0
        //
  };
  //
  // x0^exp0 == erg
  //
  // (Schleifenvariante = i)
  //

  return erg;

  // Problemfall: x == 0, exp == 0

};

int main()
{
 cout << setprecision(10) << power2(13.5, 3) << endl;
 cout << setprecision(10) << power2(2.0, 10) << endl;

 return 0;
}
```

Bei der Analyse mit Hilfe der „Zusicherungen“ in Kommentarform wurde der Problemfall `x == 0, exp == 0` als mathematisch nicht korrekt behandelt (und auch nicht richtig spezifiziert) erkannt! Ändern Sie die Spezifikation und das Programm.

1.3.4 Explizite und implizite Spezifikation von Funktionen

Funktionen können implizit (durch Angabe von Eigenschaften)

```
1.0  max (s : ℕ1-set) m : ℕ1
 .1  pre  card s ≠ 0
 .2  post m ∈ s ∧ ∀ x ∈ s · m ≥ x
```

oder explizit (durch Angabe eines Algorithmus) spezifiziert werden:

```
2.0  max : ℕ1-set → ℕ1
 .1  max (s) ≜
 .2     (dcl maxBisher : ℕ1 := getFirstElement(s) ;
 .3     while existsNextElement(s)
 .4     do let n = getNextElement(s)  in
 .5         if n > maxBisher then maxBisher := n;
 .6
 .7     return maxBisher
 .8     )
 .9  pre  card s ≠ 0
```

Diskutieren sie Vor- und Nachteile.

Eine explizite Spezifikation von *max*() in C++ fehlt hier noch, da wir die Datenstruktur `set` erst in Kapitel 2 kennenlernen werden.

Bemerkung: Für Containerdatentypen definierte Methoden wie

getFirstElement(), getNextElement(), existsNextElement(),

die im Sinne obiger expliziter Spezifikation bei der Benutzung in einer Schleife dazu dienen, jedes Element des Containerdatentyps genau einmal zu bearbeiten, nennt man **Iteratoren**.

1.3.5 Spezifikation von Bausteinen von Softwaresystemen

Die vordefinierte Klasse `<string>`:

Die Klasse `<string>` besitzt folgende Methoden:

Konstruktoren	`string()`
	`string(const char*)`
	`string(const string&)`
	`string(const string&, size_type, size_type)`
Destruktor	`~string()`
Klassenmethoden (member functions)	`int length()`
	`string& operator= (const string& s)`
	`string& operator= (const char* p)`
	`string& operator+= (const string& s)`
	⋮
	`char& operator[](int)`
	`const char& operator[](int) const`
	⋮
friend-Funktionen:	`ostream& operator << (ostream &o, const string& s);`
	`istream& operator >> (istream &i, string &s);`
	`bool operator== (const string& s1, const string& s2)`
	⋮

Die Anweisung `string s2(s1, 8, 2);` legt ein neues Exemplar des Datentyps `string` namens `s2` an und initialisiert dessen Wert mit einem Teilstring von `s1`, nämlich demjenigen ab Zeichenposition 8 und mit der Länge 2.

`string` ist ein Prototyp für sogenannte Containertypen, hier für eine angeordnete Sammlung — d.h. eine Folge — von `char`s.

Ein Beispiel zur Textbearbeitung mit Hilfe von `string`-Methoden:

```
/////////////////////////////////
// Datei:   satz.cc
// Version: 0.9
// Zweck:   string demo
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
/////////////////////////////////

#include    <iostream>
#include    <string>

using namespace std;

int main()
{
  string Satz;

  Satz = "Eskimos beschreiben Schnee auf ";
  Satz += "23 verschiedene Weisen";

  int pos(0);

  if (pos < Satz.length()-1) {
        do {
            cout << Satz[pos];
            ++pos;
        } while (pos < Satz.length());

        cout << endl;
  };

  int anz_e(0);
  for (int j=0; j < Satz.length(); j++)
        if (Satz[j] == 'e') anz_e++;
  cout << anz_e << " e-Vorkommnisse" << endl;

  return 0;
}
```

Aufgaben:

a) Testen Sie.

b) Schreiben Sie eine Variante, die alle vorkommenden Buchstaben zählt und eine Statistik ausgibt.

Eigentlich ist `string` ein template-Typ, der mit dem Komponententyp `char` vorinstantiiert ist.

Zugriff auf `string`-Elemente (`char`'s) über Index:

	Index
akt. Position initialisieren	`int i(0);`
akt. Position verschieben	`i++;`
Abbruch	`i < x.length()`
Element an akt. Stelle	`x[i]`

Eine eigene einfache Klasse: Sparbuch

```
Sparbuch
----------------------------------------------------------
void zahleEin(DM Betrag);
void hebeAb(DM Betrag);
void schreibeZinsenGut();
void aktualisiere Sparbuch();
DM zeigeGuthaben();
void setzeNeuenZinssatz(double Zinssatz, const Datum& d);
void legeSparbuchAn());
DM loeseSparbuchAuf();
void druckeSparbuch();
bool saveSparbuch();
bool openSparbuch();
...
```

mit

```
DM
----------------------------------
- double Wert;
----------------------------------
DM(double w);
double zeigeWert();
void setzeWert(const Euro& ew);
...
```

```
Euro
----------------------------------
- double Wert;
----------------------------------
Euro(double w);
double zeigeWert();
void setzeWert(const DM& dw);
...
```

Datum
- Jahr j; - Monat m; - Tag t;

Datum(const Jahr& jj, const Monat& mm, const Tag& tt);
void druckeDatum();
void inkrementiereDatum();
...

und weiteren Hilfsklassen.

Eine explizite Spezifikation der beiden Klassen `DM` und `Euro` in Form von C++-Code, die die automatische Konvertierung von `DM` in `Euro` verdeutlicht, sehen Sie hier:

```
/////////////////////////////////////////
// Datei:   DM_Euro.cc
// Version: 1.1
// Zweck:   DM und Euro
// Autor:   Holger Arndt
// Datum: 23.05.2001
/////////////////////////////////////////

#include <iostream>
#include <iomanip>

using namespace std;

class DM;

class Euro
{
private:
  double Wert;
public:
  Euro() : Wert(0.0) {};
  Euro(double w) : Wert(w) {};
  Euro(const Euro &e) : Wert(e.Wert) {};
  Euro(DM dw);
  double ZeigeWert() const { return Wert; };
};

class DM
{
private:
  double Wert;
public:
```

```
  DM() : Wert(0.0) {};
  DM(double w) : Wert(w) {};
  DM(const DM &d) : Wert(d.Wert) {};
  DM(Euro ew) : Wert(ew.ZeigeWert() * 1.95583) {};
  double ZeigeWert() const { return Wert; };
};

Euro::Euro(DM dw)
{
  Wert = dw.ZeigeWert() / 1.95583;
}

void DruckeEuroBetrag(const Euro &e)
{
  cout << "Geldbetrag: " << setiosflags(ios::fixed) << setprecision(2)
       << e.ZeigeWert() << " Euro" << endl;
}

int main()
{
  Euro b1(12.3);
  Euro b2(14.12);
  DM b3(1.23);
  Euro b4;
  Euro b5(b1);

  DruckeEuroBetrag(b1);
  DruckeEuroBetrag(b2);
  DruckeEuroBetrag(b3);
  DruckeEuroBetrag(b4);
  DruckeEuroBetrag(b5);

  return 0;
}
```

Um Qualitätsprobleme von Software zu vermeiden, ist es angebracht, statt mit `double, int, ...` besser mit Klassen, die Einheiten realisieren (z.B.: `DM, kg, m, mm, ...`), zu arbeiten. Entweder können dann automatische Konversionen oder zumindest Compilerfehlermeldungen vor Einheitenvermischungen schützen.

Index

DruckeEuroBetrag, 17

Euro::Euro, 17

main, 14, 17

ZeigeWert, 16, 17