

Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2015

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München



Programm heute

- ① Einführung
- ② Grundlagen von Algorithmen
- ③ Grundlagen von Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
 - Motivation und Spezifikation
 - Verifikation
 - Beispiel: Insertion Sort
 - Validation

Beispiele für Fehler I

- 1996: Explosion der Ariane 5 Rakete mit 4 Satelliten an Bord
 - **Schaden:** ca. 500 Millionen Dollar
 - **Ursache:** Wert der horizontalen Geschwindigkeit von 64 Bit Floating Point Zahl (`double`) konvertiert auf 16 Bit Ganzzahl (`signed short`)
 - Zahl grösser als $2^{15} = 32768$ → **overflow**
 - Zusammenbruch des Lenksystems
 - Backup-Rechner verwendet gleiches Programm, selber Fehler
 - Selbstzerstörung nach 36.7 Sekunden Flugzeit
- Annahme dass Rakete nie so schnell fliegen wird...

Beispiele für Fehler II

- 1995: Ausfall Stellwerk Hamburg-Altona
 - **Schaden:** 2 Tage kein Schienenverkehr in Hamburg-Altona, 2 Monate Notfahrplan
 - **Ursache:** Stack-Overflow
 - neues zentrales, computerisiertes Stellwerk für 62.6 Millionen D-Mark von Siemens
 - 3.5kB Stack für Stellbefehle (als sequentielle Liste)
 - zu klein schon für normalen Verkehr → **overflow**
 - Fehlerbehandlungsroutine fehlerhaft → Endlosschleife

Beispiele für Fehler III

- 1985 – 1987: Therac-25 Maschine zur Strahlentherapie
 - **Schaden:** 3 Todesfälle durch Strahlenüberdosis
 - **Ursache:** “race condition” in Software-Sperre für Strahl
(**race condition:** Ergebnis einer Operation hängt vom zeitlichen Verhalten der Einzeloperationen ab, siehe Multitasking)
 - **Overflow** in 8 Bit Test-Zähler zusammen mit Benutzereingabe deaktivieren Software-Sperre für Strahl
 - unkontrollierte Bestrahlung mit bis zu 100-facher Dosis
 - 40 - 200 Gray Dosis statt 2 Gray, ab 10 Gray tödlich
 - mind. 3 Personen sterben an Strahlenüberdosis

Beispiele für Fehler IV

- 2011 – 2014: Heartbleed Fehler in OpenSSL
 - **Schaden:** unklar (potentiell Millionen von kompromittierten Accounts)
 - **Ursache:** Buffer-Overflow in verschlüsselter Kommunikation
 - keine Prüfung der Grösse eines angeforderten Arrays
 - durch eigene Speicherverwaltung in OpenSSL gelangen sensitive Daten an potentielle Angreifer (Passwörter, Zertifikate)
 - viele Webservices betroffen (von Amazon bis Wikipedia)
 - viele Geräte betroffen (von Android Jelly Bean bis Cisco Router)

Relative Korrektheit

Die **Korrektheit** eines Algorithmus bezieht sich immer auf die **Spezifikation** dessen, was er tun **soll**.

- Korrektheit ist **relativ**
- Algorithmus kann nicht an sich als korrekt bewiesen werden, sondern nur in Bezug auf Spezifikation

Spezifikation: eindeutige Festlegung der berechneten Funktion bzw. des Terminierungsverhaltens einer Softwarekomponente

Nachweis der Korrektheit

Nachweis der Korrektheit durch

- **Verifikation:** formaler, mathematischer Beweis, dass Algorithmus korrekt bezüglich formaler Spezifikation
 - erfordert Formalisierung der Algorithmensprache und Spezifikation
- **Validation:** nicht-formaler Nachweis der Korrektheit etwa durch systematisches Testen
 - in der Regel kein 100%ig sicherer Nachweis der Korrektheit!

Verifikation vs. Validation

E.W. Dijkstra (1930 - 2002):

“Testing shows the presence, not the absence of bugs”

Analogie zur Mathematik:

- **Spezifikation:** entspricht Satz
- **Verifikation:** Beweis des Satzes
- **Validation:** Ausprobieren an Beispielwerten

Programm heute

- ① Einführung
- ② Grundlagen von Algorithmen
- ③ Grundlagen von Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
 - Motivation und Spezifikation
 - Verifikation**
 - Beispiel: Insertion Sort
 - Validation

Vor- und Nachbedingungen

Vor- und Nachbedingungen: Methode zur Spezifikation von gewünschten Eigenschaften von Algorithmen

{VOR} ANW {NACH}

- {VOR}: Aussage über Zustand **vor** Ausführung von ANW
- {NACH}: Aussage über Zustand **nach** Ausführung von ANW

Bedeutung:

- Gilt die Vorbedingung {VOR} unmittelbar vor Ausführung von ANW
- und terminiert ANW
- so gilt die Nachbedingung {NACH} unmittelbar nach Ausführung von ANW

Vor- und Nachbedingung

{VOR} ANW {NACH}

Sonderfälle:

- terminiert ANW nicht, so ist Aussage immer wahr
 - egal wie {VOR}, {NACH} aussehen
- gilt {VOR} nicht, so ist Aussage auch immer wahr
 - egal ob ANW terminiert oder ob {NACH} gilt

Vor- und Nachbedingung: Beispiele I

- Aussage

$$\{X==0\} X = X+1 \{X==1\}$$

ist wahr

- Aussage

$$\{\text{true}\} X = Y \{X==Y\}$$

ist wahr

- Aussage

$$\{Y==a\} X = Y \{X==a \wedge Y==a\}$$

ist wahr für alle $a \in \mathbb{Z}$

Vor- und Nachbedingung: Beispiele II

- Aussage

$$\{X==a \wedge Y==b \wedge a \neq b\} X = Y; Y = X \{X==b \wedge Y==a\}$$

ist **falsch** für alle $a, b \in \mathbb{Z}$!

(nach erstem Schritt haben X, Y beide Wert b - typischer Fehler bei Wertetausch!)

- Aussage

$$\{X==a \wedge Y==b\} Z = X; X = Y; Y = Z \{X==b \wedge Y==a\}$$

ist wahr für alle $a, b \in \mathbb{Z}$ (korrekter Wertetausch mit Hilfsvariable Z)

Vor- und Nachbedingung: Beispiele III

- Aussage

$\{\text{false}\}$ ANW $\{\text{NACH}\}$

ist wahr für alle ANW und NACH, da Vorbedingung falsch

- Aussage

$\{\text{true}\}$ ANW $\{\text{false}\}$

ist genau dann wahr, wenn ANW nicht terminiert

- Aussage

$\{X==a\}$ **while** $X \neq 0$ $\{ X = X-1 \}$ $\{X==0\}$

ist wahr für alle $a \in \mathbb{Z}$

(auch für $a < 0$, da dann while-Schleife nicht terminiert!)

Partielle Korrektheit

Partielle Korrektheit

Ein Programm mit Anweisungen ANW sowie Vorbedingung $\{VOR\}$ und Nachbedingung $\{NACH\}$ heißt **partiell korrekt** bezüglich VOR, NACH genau dann, wenn

$$\{VOR\} \text{ ANW } \{NACH\}$$

wahr ist.

Das Programm heißt **total korrekt** bezüglich VOR, NACH genau dann, wenn

- es partiell korrekt ist bezüglich VOR, NACH ist und
- ANW immer dann terminiert, wenn VOR gilt.

Bausteine von Algorithmen

Wiederholung: Bausteine von Algorithmen

- Elementarer Verarbeitungsschritt (z.B. Zuweisung an Variable)
- Sequenz (elementare Schritte nacheinander)
- Bedingter Verarbeitungsschritt (z.B. if/else)
- Wiederholung (z.B. while-Schleife)

Korrektheit von Anweisungstypen I

- Elementarer Verarbeitungsschritt α

$$\{\text{VOR}\} \alpha \{\text{NACH}\}$$

- Sequenz $\alpha; \beta$

$$\{\text{VOR}\} \alpha; \beta \{\text{NACH}\}$$

wird gezeigt mittels Zwischenbedingung MITTE, also

$$\{\text{VOR}\} \alpha \{\text{MITTE}\}$$

und

$$\{\text{MITTE}\} \beta \{\text{NACH}\}$$

Korrektheit von Anweisungstypen II

- Bedingter Verarbeitungsschritt

$\{\text{VOR}\} \text{ if } (B) \{ \alpha \} \text{ else } \{ \beta \} \{\text{NACH}\}$

wird gezeigt mittels

$\{\text{VOR} \wedge B\} \alpha \{\text{NACH}\}$

und

$\{\text{VOR} \wedge \neg B\} \beta \{\text{NACH}\}$

→ zeige also dass NACH gilt, egal welcher Zweig ausgeführt wird!

Korrektheit von Anweisungstypen III

- Wiederholung

{VOR}
 while (B) { β }
{NACH}

wird gezeigt mittels **Schleifeninvariante P**:

- 1 prüfe, daß $VOR \Rightarrow P$
(Vorbedingung garantiert Schleifeninvariante bei Eintritt in Schleife)

- 2 prüfe, daß

$\{P \wedge B\} \beta \{P\}$

(wenn Schleife durchlaufen, bleibt Schleifeninvariante wahr)

- 3 prüfe, daß $\{P \wedge \neg B\} \Rightarrow NACH$
(Nachbedingung muß nach Verlassen der Schleife gelten)

Verifikation: Beispiel I

Gauss'sche Summenformel

Sei $n \in \mathbb{N}$. Dann gilt

$$1 + 2 + \dots + n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Algorithmus:

```
{ n > 0 }  
  sum = 0;  
  i = 0;  
  while (i < n) {  
    i = i + 1;  
    sum = sum + i;  
  }  
{ sum == n*(n+1)/2 }
```

Verifikation: Beispiel II

Sequenz mit Zwischenbedingung:

```
{ n > 0 }  
  sum = 0;  
{ n > 0 ∧ sum == 0 }  
  i = 0;  
{ n > 0 ∧ sum == 0 ∧ i == 0 }
```

Algorithmus:

```
{ n > 0 }  
  sum = 0;  
  i = 0;  
  while (i < n) {  
    i = i + 1;  
    sum = sum + i;  
  }  
{ sum == n*(n+1)/2 }
```

Verifikation: Beispiel III

Wiederholung mit **Invariante**:

```
{ n > 0 ∧ sum == 0 ∧ i == 0 }  
  while (i < n) {  
    i = i + 1;  
    sum = sum + i;  
  }  
{ sum == n*(n+1)/2 }
```

Passende Invariante P?

$$P = (\text{sum} == i*(i+1)/2 \wedge i \leq n)$$

Algorithmus:

```
{ n > 0 }  
  sum = 0;  
  i = 0;  
  while (i < n) {  
    i = i + 1;  
    sum = sum + i;  
  }  
{ sum == n*(n+1)/2 }
```

Verifikation: Beispiel IV

Prüfe:

① $\{\text{VOR}\} \Rightarrow \{\text{P}\}$

es ist $i==0$, also $i \leq n$, und
 $\text{sum} == 0 * 1 / 2 == 0$ ✓

② $\{\text{P} \wedge \text{B}\} \beta \{\text{P}\}$

es ist $i < n$, also nach $i=i+1$ immer
noch $i \leq n$, und für sum gilt

$$\text{sum} = \frac{i(i+1)}{2} + (i+1) == \frac{(i+1)*(i+2)}{2} \quad \checkmark$$

③ $\{\text{P} \wedge \neg \text{B}\} \Rightarrow \{\text{NACH}\}$

es ist $\text{P} \wedge \neg \text{B} \Leftrightarrow \text{P} \wedge !(i < n)$

$$\Leftrightarrow \text{sum} == \frac{i(i+1)}{2} \wedge i \leq n \wedge !(i < n)$$

$$\Leftrightarrow \text{sum} == \frac{i(i+1)}{2} \wedge i == n$$

$$\Rightarrow \text{sum} == \frac{n(n+1)}{2} \quad \checkmark$$

Wiederholung mit **Invariante**:

$$\{ n > 0 \wedge \text{sum} == 0 \wedge i == 0 \}$$

```
while (i < n) {  
    i = i + 1;  
    sum = sum + i;  
}
```

$$\{ \text{sum} == n*(n+1)/2 \}$$

Invariante P:

$$P = (\text{sum} == i*(i+1)/2 \wedge i \leq n)$$

Verifikation

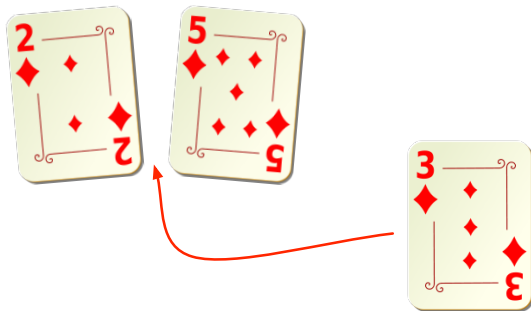
- Verifikation ist manuell sehr aufwendig
 - in Praxis daher leider sehr selten eingesetzt
- Vorgang lässt sich weitgehend automatisieren
 - Programm-Verifizierer
 - Programm mit Vor- und Nachbedingungen
 - Schleifen müssen Invarianten haben
 - nur möglich mit eingeschränkten Programmiersprachen
z.B. Lightweight Java
 - an der TUM: Programmsystem Isabelle (Informatik)
<http://isabelle.in.tum.de>

Programm heute

- ① Einführung
- ② Grundlagen von Algorithmen
- ③ Grundlagen von Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
 - Motivation und Spezifikation
 - Verifikation
 - Beispiel: Insertion Sort**
 - Validation

Insertion Sort

Insertion Sort: Sortieren durch direktes Einfügen



Pseudocode von Insertion Sort

Input: Feld $A[0..n-1]$ von n natürlichen Zahlen

Output: Feld A aufsteigend sortiert

InsertionSort(A):

```
for j=1 to länge(A)-1 {
    key = A[j];
    // füge A[j] in sortierte Liste A[0..j-1] ein
    i = j-1;
    while (i >= 0 && A[i] > key) {
        A[i+1] = A[i];
        i = i-1;
    }
    A[i+1] = key;
}
```

Insertion Sort: Ablauf

Beispielablauf:

Eingabe: $A = 6, 3, 5, 7, 2, 4$

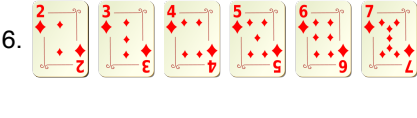
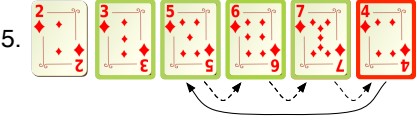
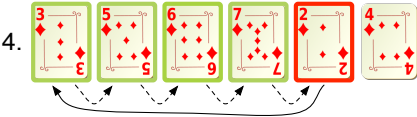
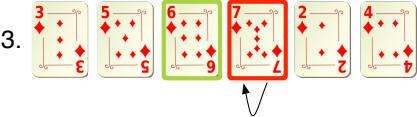
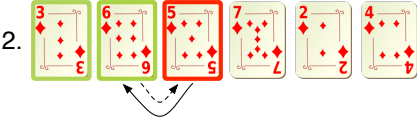
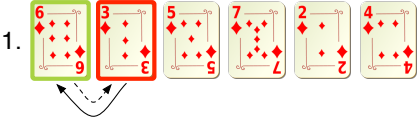
j					
key					
i					
$A[0..j-1]$					

4	2	7	5	3	6
---	---	---	---	---	---

InsertionSort(A):

```
for j=1 to länge(A)-1 {  
    key = A[j];  
    // füge A[j] in sortierte  
    // Liste A[0..j-1] ein  
    i = j-1;  
    while (i >= 0 && A[i] > key) {  
        A[i+1] = A[i];  
        i = i-1;  
    }  
    A[i+1] = key;  
}
```

Insertion Sort: Ablauf



Insertion Sort: Verifikation

Entscheidendes Kriterium: **Schleifen-Invariante P** der **for** Schleife

$P =$ Teilfeld $A[0..j-1]$ besteht aus den ursprünglich in $A[0..j-1]$ enthaltenen Werten und $A[0..j-1]$ ist geordnet

Prüfe:

- 1 Anfang $j=1$: Feld $A[0..j-1]$ enthält nur ein Element $A[0]$, ist sortiert ✓
- 2 aktuelles Element mit Wert key , die Werte $A[j-1]$, $A[j-2]$, ... werden jeweils eine Position nach rechts verschoben, bis korrekte Position $A[i]$ gefunden für aktuelles Element, Element wird dort eingefügt → Feld $A[0..j-1]$ sortiert ✓
- 3 Algorithmus terminiert wenn $j==n$, also gilt mit voriger Invariante P : $A[0..n-1]$ ist sortiert ✓

Programm heute

- ① Einführung
- ② Grundlagen von Algorithmen
- ③ Grundlagen von Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
 - Motivation und Spezifikation
 - Verifikation
 - Beispiel: Insertion Sort
 - Validation**

Validation

- **Validation:** nicht-formaler Nachweis der Korrektheit, etwa durch systematisches Testen

Warum **Validation** wenn wir **Verifikation** haben?

- bei Verifikation können Fehler unterlaufen sein
- Verifikation zu aufwendig oder nicht mehr möglich für größere bzw. komplexe Programme
- der verwendete Compiler/Rechner kann fehlerhaft sein
- **aber!** Testen kann nur Anwesenheit von Fehlern zeigen, nicht die Abwesenheit!

Fehlerarten

Mögliche Fehlerarten (nur ein Auszug!)

- Denkfehler bei der Konstruktion des Algorithmus
 - z.B. Annahme falscher Rahmenbedingungen
 - oder Problemstellung missverstanden
- Rechenfehler bei der Verifikation
 - z.B. bei Invarianten-Prüfung falsche Schlussfolgerung gezogen
- Tipp- oder Programmierfehler bei Umsetzung auf Computer
 - z.B. Index-Zählung bei 1 statt 0 angefangen
 - oder Ziffer 1 statt Buchstabe l getippt
- Fehler im Compiler, Betriebssystem oder Rechner selbst
 - kein Compiler ist fehlerfrei
 - kein Betriebssystem ist fehlerfrei
 - kein Rechner ist fehlerfrei

Validation durch systematische Tests

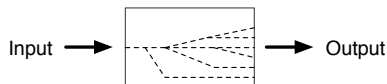
- **Systematische Tests:** systematische “Jagd” auf mögliche Fehler im Programm
- verschiedene Test-Typen stehen zur Verfügung:
 - Blackbox-Test
 - Whitebox-Test
 - Regressions-Test
 - Integrations-Test

Blackbox-Test



- Programm als “Blackbox” betrachten
 - interne Struktur unbekannt
 - nur Eingabe und Ausgabe bekannt
 - sowie Dokumentation was als Eingabe erlaubt ist
- systematischer Test auf mögliche Eingabedaten (und ob korrektes Ergebnis geliefert wird) → datenbezogenes Testen
- Repräsentative Werteanalyse:
 - Klassen von Eingabedaten (z.B. negative ganze Zahlen)
 - Auswahl von Repräsentanten dieser Klasse zum Test
- Grenzwertanalyse:
 - Eingabedaten haben meist Grenzwerte (z.B. Zahlen von -128 bis 127)
 - direkte Tests auf die Grenzwerte und auch darüber hinaus

Whitebox-Test

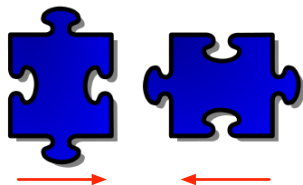


- Programm als “Whitebox” betrachten
 - innere Struktur bekannt
- systematischer Test auf **innere Struktur** des Programms, d.h. das alle Programmteile ausgeführt werden → ablaufbezogenes Testen
- Test-Varianten (Auszug):
 - Ausführung **sämtlicher Programmwege** inkl. Kombinationen (meist unpraktikabel!)
 - **Alle Schleifen** müssen nicht nur einmal, sondern **zweimal** durchlaufen werden
 - **Alle Anweisungen** sollen mindestens **einmal** ausgeführt werden

Regressions-Test

- **Annahme:** mehrere Versionen des Programms (mit gleicher Funktionalität!) sind verfügbar
- systematischer Test über längere Zeit der **verschiedenen Versionen mit gleicher Eingabe**
 - ist Ausgabe unterschiedlich, ist eine der Versionen fehlerhaft
- insbesondere zur Aufdeckung von bereits behobenen Fehlern, die wieder auftauchen (sog. **“Regressionen”**)

Integrations-Test



- große Softwaresysteme bestehen aus **Modulen** (natürlich separat getestet!)
- systematischer Test der **Zusammenarbeit** der Module, schrittweise zusammengefügt zum Gesamtsystem
 - z.B. mittels Blackbox- und Whitebox-Tests

Testen in der Praxis

- “poor man’s testing” mit `assert` in C++ (siehe `assert.h`)
 - für Debug-Modus:
`assert(length >= 0 && "oops, negative Länge geht nicht");`
 - Prüfung deaktiviert im Release-Modus mittels Makro `NDEBUG`
- hiermit läßt sich auch auf Vor- und Nachbedingungen testen
- **Unit Test Frameworks:** Gerüste/Systeme zur Vereinfachung der Implementation von Tests
 - z.B. Microsoft Unit Testing Framework in Visual Studio
 - oder Google C++ Testing Framework
 - oder Boost Unit Testing Framework

Fehler in der Praxis

- **fehlertolerantes** Programmieren
 - Fehler direkt im Programm feststellen
 - möglicherweise sogar korrigieren
 - Beispiel: “keine Verbindung zum Internet, stellen Sie zuerst Internet-Verbindung her, bevor Programm weiter ausgeführt werden kann”
- **fehlerpräventives** Programmieren
 - Verwendung geeigneter Programmiersprache mit statischem Typsystem (z.B. C++)
 - Meiden von expliziter Typumwandlung (type casts)
 - Verwendung von sicheren Konstrukten (z.B. Smartpointer statt Zeiger)
 - Aktivierung von allen Compiler-Warnungen (z.B. Clang ist hier besonders hilfreich)
- Tests bereits in Spezifikations-Phase entwickeln (**test-driven development**)

Zusammenfassung

- ① Einführung
- ② Grundlagen von Algorithmen
- ③ Grundlagen von Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
 - Motivation und Spezifikation
 - Verifikation
 - Beispiel: Insertion Sort
 - Validation