

# Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Dr. Tobias Lasser

Computer Aided Medical Procedures  
Technische Universität München



# Programm heute

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

# 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

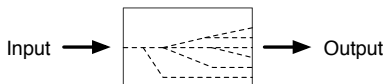
- **Systematische Tests:** systematische “Jagd” auf mögliche Fehler im Programm
- verschiedene Test-Typen stehen zur Verfügung:
  - Blackbox-Test
  - Whitebox-Test
  - Regressions-Test
  - Integrations-Test

# 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



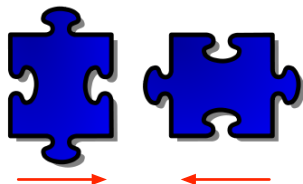
- Programm als “Whitebox” betrachten
  - innere Struktur bekannt
- systematischer Test auf **innere Struktur** des Programms, d.h. das alle Programmteile ausgeführt werden → ablaufbezogenes Testen
- Test-Varianten (Auszug):
  - Ausführung **sämtlicher Programmwege** inkl. Kombinationen (meist unpraktikabel!)
  - **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, 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**)

# Programm heute

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

# Wozu Effizienz?

“Wenn das Programm zu langsam läuft, kaufen wir einfach einen schnelleren Rechner!”

- der schnellere Rechner ist eventuell trotzdem nicht schnell genug
- auch wenn es schnell genug läuft, Energie sparen (z.B. wegen Akkulaufzeit!)

# Komplexität von Algorithmen

Interessante Fragen bei **Algorithmus A**:

- wie viel **Speicherplatz** benötigt Algorithmus?
- wie viel **Rechenzeit** benötigt Algorithmus?

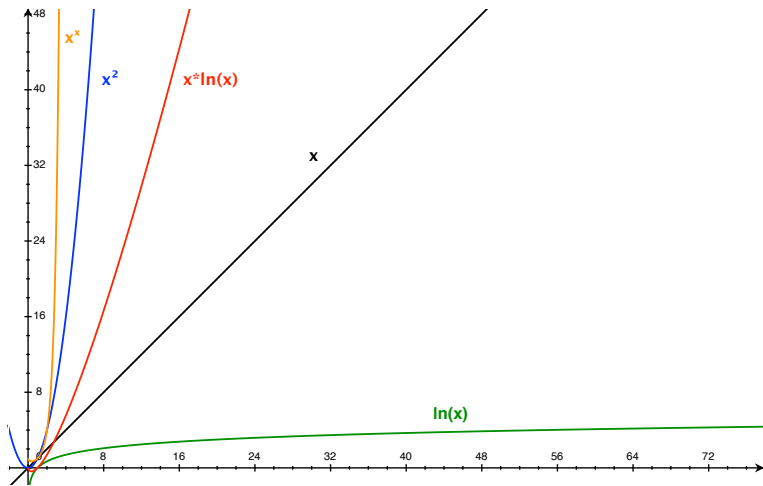
→ **Komplexitätsanalyse**

Wesentlicher Faktor bei Komplexität: **Größe der Eingabe n**

- benötigter Speicher abhängig von n
- benötigte Rechenzeit abhängig von n

→ Funktion  $f$  zur Beschreibung des **Wachstumsverhaltens** von A

# Typische Wachstumsraten



## Wiederholung: Insertion Sort

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

**Output:** Feld  $A$  aufsteigend sortiert

**InsertionSort(A):**

```
1  for j=1 to länge(A)-1 {
2    key = A[j];
3    // füge A[j] in sortierte Liste A[0..j-1] ein
4    i = j-1;
5    while (i >= 0 && A[i] > key) {
6      A[i+1] = A[i];
7      i = i-1;
8    }
9    A[i+1] = key;
10 }
```



# Komplexität von Insertion Sort

Speicher:

- Feld  $A$  (Länge  $n$ ), eine extra Variable (key)

Zeit:

- Annahme: jeder elementare Verarbeitungsschritt in Zeile  $i$  benötigt konstante Zeit  $c_i \in \mathbb{R}$
- Beschreibe Laufzeit mittels

Laufzeitfunktion  $T : \mathbb{N} \rightarrow \mathbb{R}$ ,  $T : n \mapsto T(n)$

- $T(n)$  beschreibt Wachstumsverhalten in Abhängigkeit von Eingabegröße  $n$

# Laufzeit Insertion Sort I

## Kostenübersicht:

Zeile	Kosten	wie oft?
1	$c_1$	
2	$c_2$	
3	0	Kommentar
4	$c_4$	
5	$c_5$	
6	$c_6$	
7	$c_7$	
8	0	Blockabschluß
9	$c_9$	
10	0	Blockabschluß

## InsertionSort(A):

```
1 for j=1 to länge(A)-1 {
2   key = A[j];
3   // füge A[j] in sortierte
   // Liste A[0..j-1] ein
4   i = j-1;
5   while (i >= 0 && A[i] > key) {
6     A[i+1] = A[i];
7     i = i-1;
8   }
9   A[i+1] = key;
10 }
```

$t_j$  bezeichnet Anzahl der Abfragen der while-Bedingung in Zeile 5 für Durchlauf  $j$

# Laufzeit Insertion Sort II

Berechnung der Laufzeit  $T(n)$ :

- $T(n)$  ist Summe aller Laufzeiten der ausgeführten elementaren Verarbeitungsschritte
- Beispiel: Schritt 1 (Zeile 1) kostet  $c_1$  und wird  $n$  mal ausgeführt

$$\begin{aligned}T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) \\ &\quad + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1} (t_j - 1) \\ &\quad + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_9(n - 1)\end{aligned}$$

- Beobachtung:  $T(n)$  hängt stark von tatsächlichen Eingabe ab!

## Laufzeit Insertion Sort III

Laufzeit  $T(n)$  im **besten Fall** (best case)?

- best case: Feld A ist **bereits sortiert** bevor Algorithmus aufgerufen wird
- in while-Schleife (Zeile 5) wird Bedingung nur **einmal** abgefragt
  - es ist nämlich  $A[i] \leq \text{key}$
  - also  $t_j = 1$  für  $j = 1, \dots, n - 1$
- Also:

$$\begin{aligned}T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) \\ &\quad + c_5(n - 1) + c_6 \cdot 0 \\ &\quad + c_7 \cdot 0 + c_9(n - 1) \\ &= \underbrace{(c_1 + c_2 + c_4 + c_5 + c_9)}_{=:a} n - \underbrace{(c_2 + c_4 + c_5 + c_9)}_{=:b} \\ &= an - b \qquad (a, b \text{ Konstanten})\end{aligned}$$

## Laufzeit Insertion Sort IV

Laufzeit  $T(n)$  im **schlechtesten Fall** (worst case)?

- worst case: Feld A ist bereits **absteigend sortiert** bevor Algorithmus aufgerufen wird
- in while-Schleife (Zeile 5) muß mit **allen** Elementen  $A[0..j-1]$  verglichen werden
  - es ist nämlich  $A[i] > \text{key}$
  - also  $t_j = j + 1$  für  $j = 1, \dots, n - 1$
- Also:

$$\begin{aligned}T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) \\ &\quad + c_5 \left( \frac{n(n+1)}{2} - 1 \right) + c_6 \left( \frac{n(n-1)}{2} \right) \\ &\quad + c_7 \left( \frac{n(n-1)}{2} \right) + c_9(n - 1)\end{aligned}$$

da Gauss'sche Summenformel:

$$\sum_{j=1}^{n-1} (j + 1) = \frac{n(n+1)}{2} - 1 \quad \text{und} \quad \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

# Laufzeit Insertion Sort V

Fortsetzung worst case  $T(n)$ :

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_9(n-1) \\ &= \underbrace{\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)}_{=:a} n^2 + \underbrace{\left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9\right)}_{=:b} n \\ &\quad - \underbrace{(c_2 + c_4 + c_5 + c_9)}_{=:c} \\ &= an^2 + bn - c\end{aligned}$$

# Zusammenfassung

- ① Einführung
- ② Mathematische Grundlagen
- ③ Elementare Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
  - Motivation und Spezifikation
  - Verifikation
  - Beispiel: Insertion Sort
  - Validation
- ⑤ Grundlagen der Effizienz von Algorithmen
  - Motivation