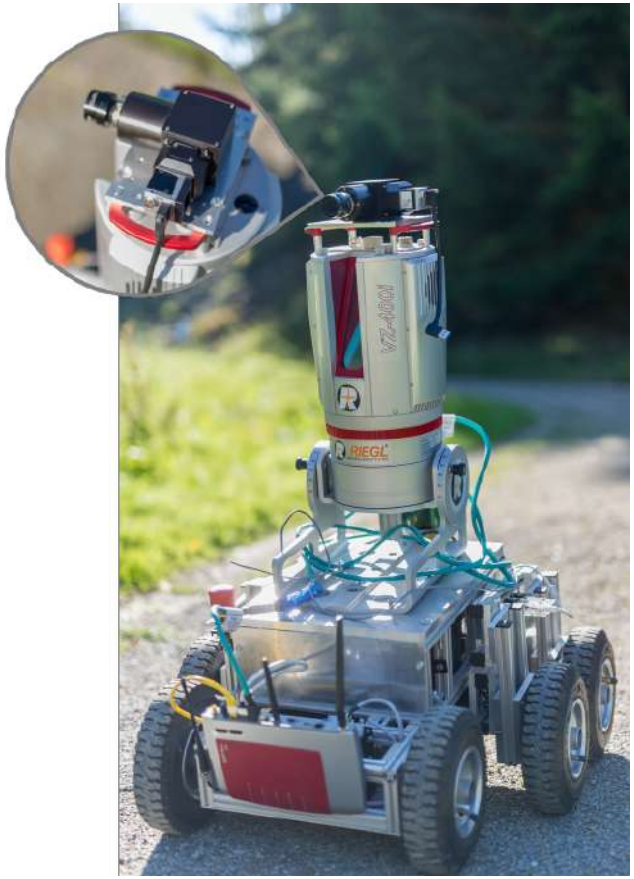
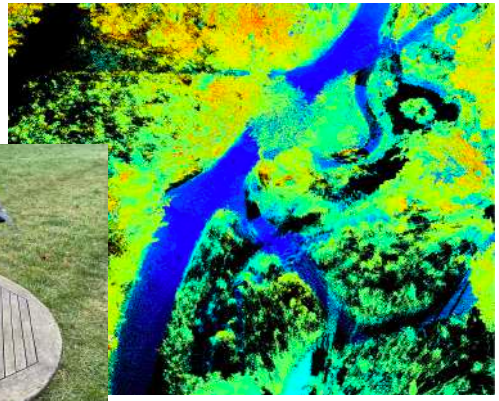
Algorithmen und Datenstrukturen

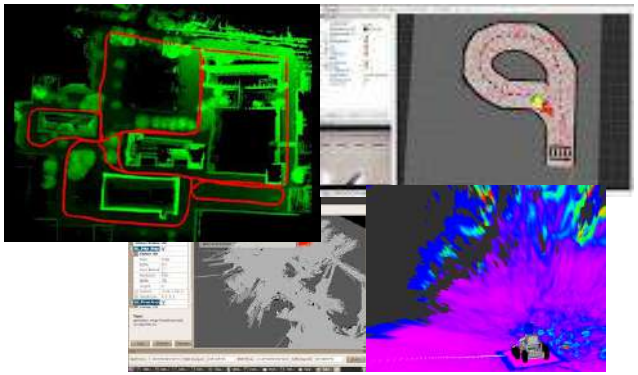
Prof. Dr. Thomas Wiemann - FB AI

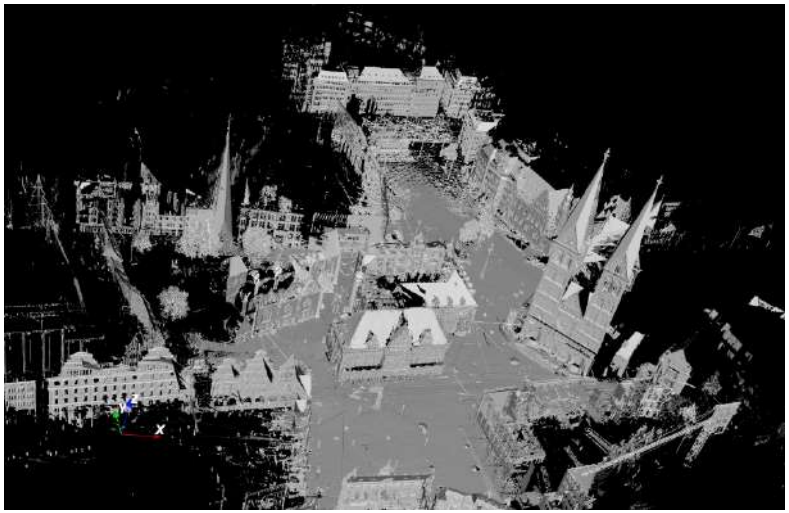
Hochschule Fulda
University of Applied Sciences

Vorstellung

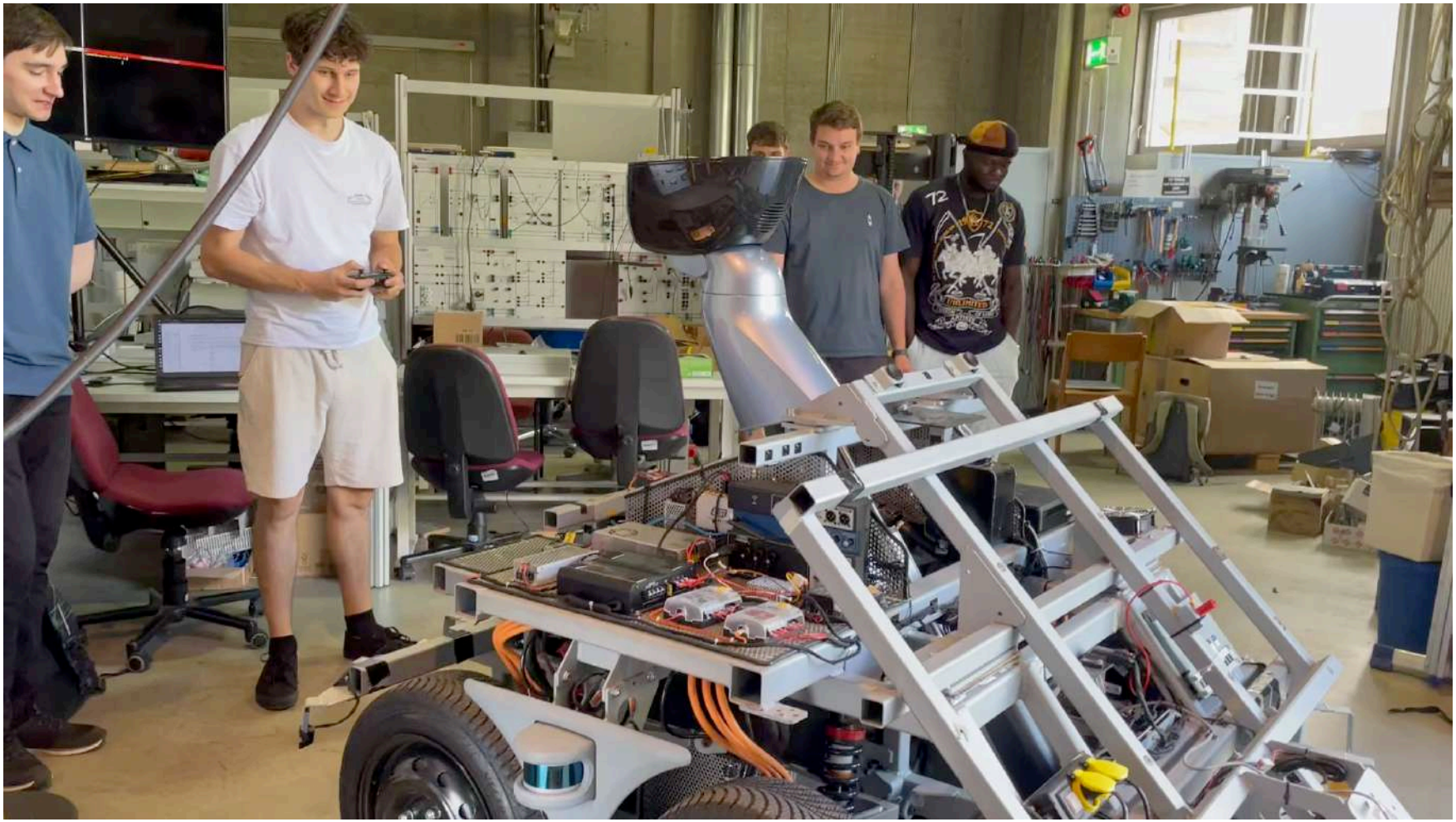




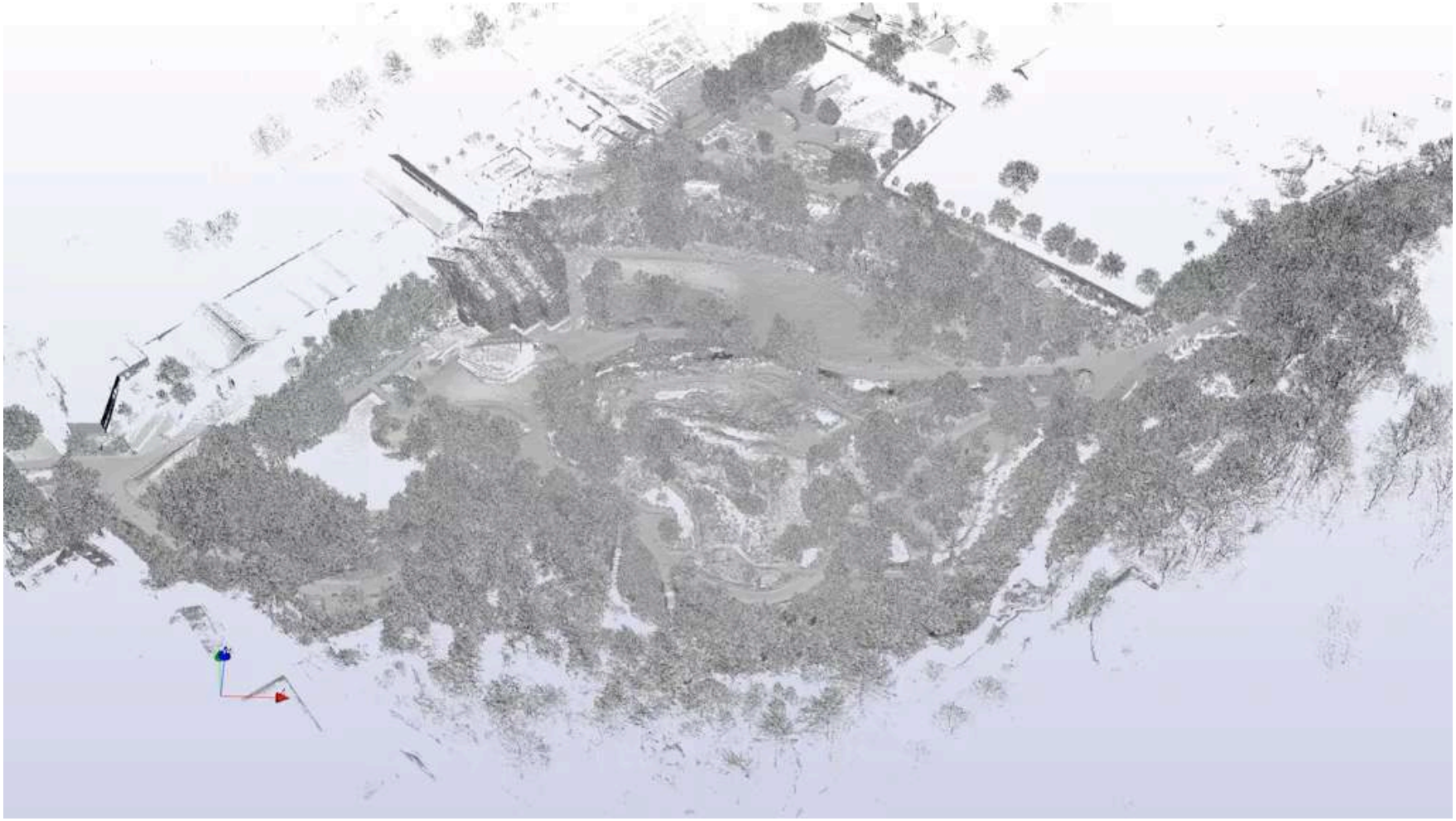




Vorstellung



Vorstellung



Vorstellung

Vorstellung

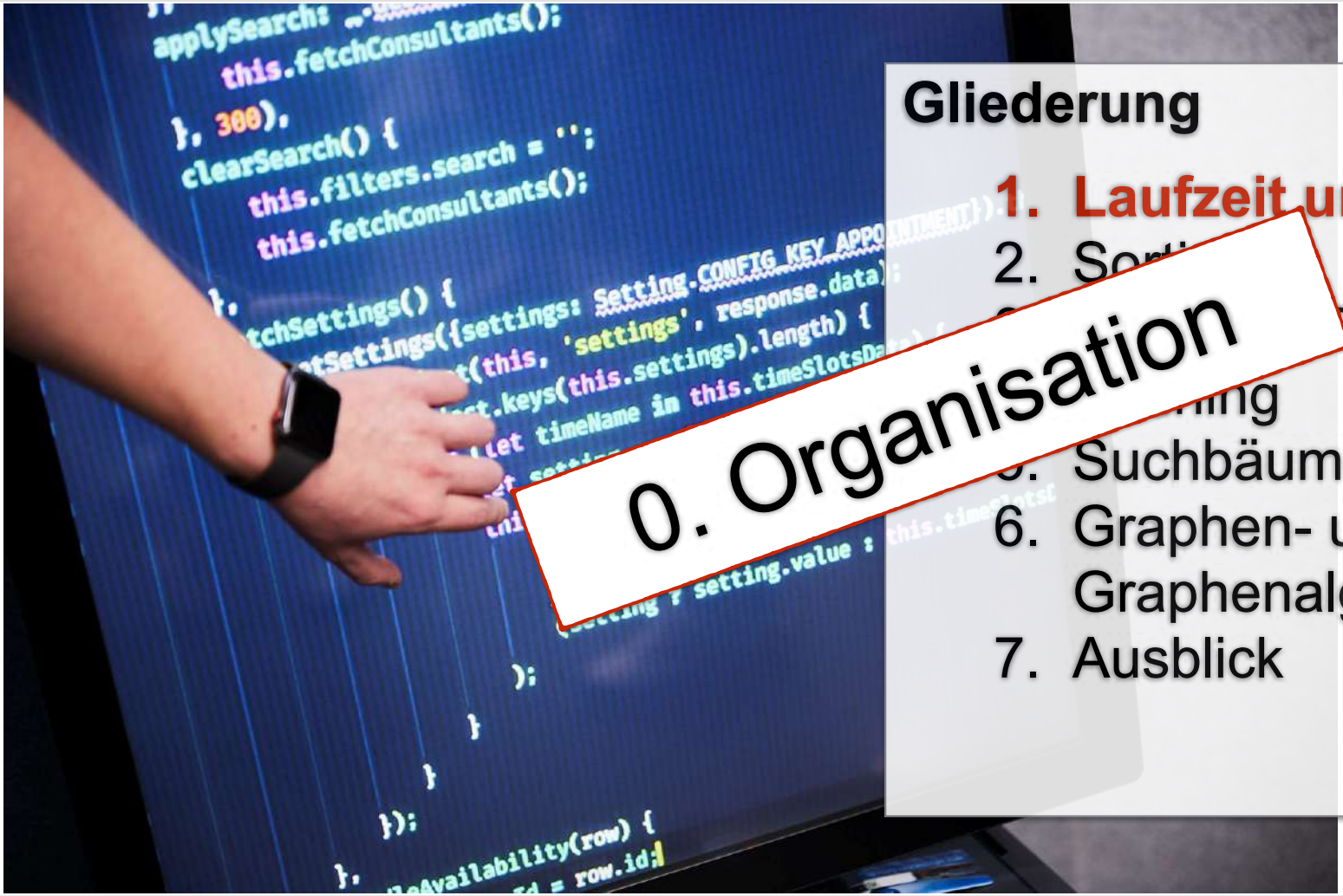
Inhalt

Gliederung

1. Laufzeit und Komplexität
2. Sortieren
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

Hochschule Fulda
University of Applied Sciences

Inhalt



Gliederung

1. Laufzeit und Komplexität
2. Sort…
3. …Datentypen
4. …ing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

0. Organisation

Organisatorisches

Kompetenzen

- Einschätzung und Messung des Laufzeitverhaltens und weiter Kosten typischer Algorithmen und Datenstrukturen
- Erweiterungen und Anpassungen von Standardimplementierungen (in Java)
- Auswahl geeigneter Standardalgorithmen und Datenstrukturen abhängig vom Einsatzszenario
- Selbständiges Erarbeiten fortgeschrittener Datenstrukturen, Bewertung des Nutzens

Portfolioprüfung*

- **Theorie:**
  - 3 Tests im Rahmen der Übungen (40%)
- **Praxis:**
  - Zweimal Vorstellung einer Hausaufgabe in der Übung (15%)
  - 2 Programmieraufgaben in 4er-Gruppen in Java mit anschließenden Testat (20%)
- **Transfer:**
  - Präsentation zu fortgeschrittenem Thema (25%)

*** Details zu den Komponenten auf den kommenden Folien**

Hochschule Fulda
University of Applied Sciences

Details zu den Portfoliokomponenten

- **Tests**
  - Drei Termine im Semester
  - Schriftliche Tests mit EvaExam (“Kreuzchenklausur”)
  - Tests werden im Rahmen der Übungen abgenommen
- **Vorträge**
  - Verteilung der Slots für die Themen der Abschlussvorträge per Moodle, Termine werden in der letzten Vorlesungswoche stattfinden
  - Themanvergabe vor den Weihnachtsferien
  - Vorträge werden in kleinen Gruppen (3 bis 4 Personen) vorbereitet und vorgetragen
  - Zeitlimit 15 min + kurze Rückfragen
  - Termin: letzte Vorlesungswoche in den bekannten Slots (VL / Übung)
- **Testate**
  - Zwei Wochen Bearbeitungszeit, Abnahme in den Übungen
  - Gleiches Bewertungsschema für alle Gruppen von TW vorgegeben
  - Ebenfalls in Kleingruppen
  - Zufällige Befragung der Gruppenmitglieder durch den Übungsleiter. Code muss erklärt werden können, ansonsten bekommt diese Person keine Punkte

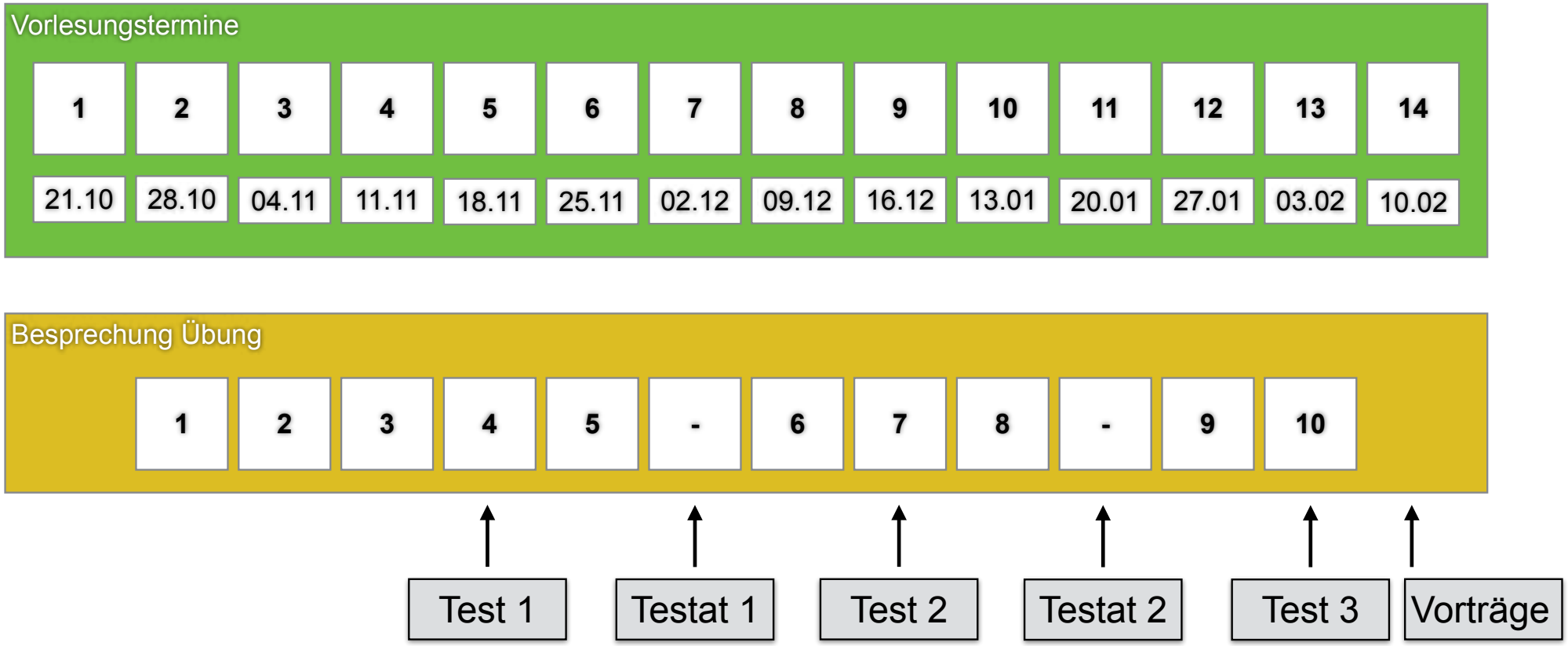
Hochschule Fulda
University of Applied Sciences

Details zu den Portfoliokomponenten

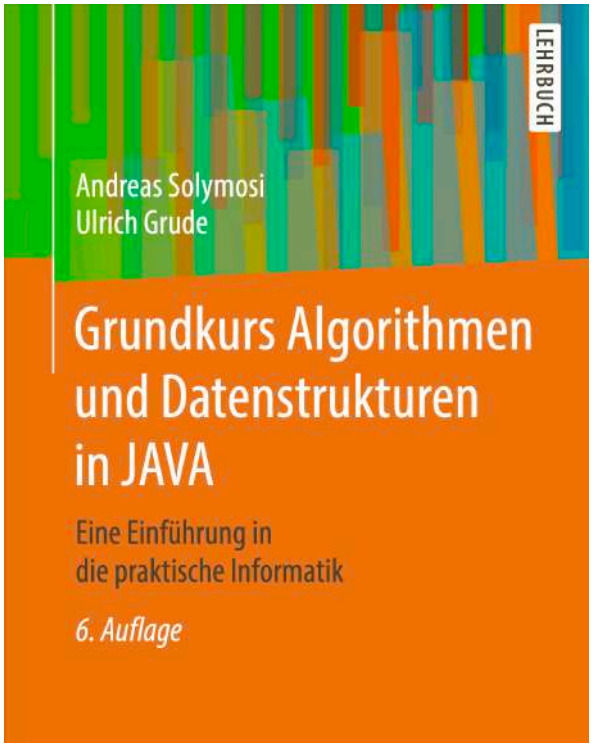
- **Vorstellung von Hausaufgaben**
  - Jede Woche (außer bei Testaten) werden in der Übung die Hausaufgaben besprochen
  - Jede*r bereitet die vor (alleine oder zu Zweit)
  - In der Sitzung wird dann für jede Aufgabe jemand (oder ein Zweierteam) bestimmt diese zu besprechen
  - Vorbereiten und Besprechen bedeutet:
    - Im Idealfall die Lösung vorzurechnen
    - Falls die Aufgabe zu Hause nicht gelöst werden konnte, soll zumindest klar werden wo die Probleme waren
    - Diese werden dann interaktiv in der Übung besprochen und aufgelöst
    - Übungsleiter*innen unterstützen
  - Drei Opt-Outs pro Person, die zu Beginn der Übung bekannt gegeben werden müssen!
  - Bei ausreichender Vorbereitung werden pro Aufgabenpräsentation 1 bis 5 Punkte vergeben, die dann anteilig in das Portfolio einfließen

Hochschule Fulda
University of Applied Sciences

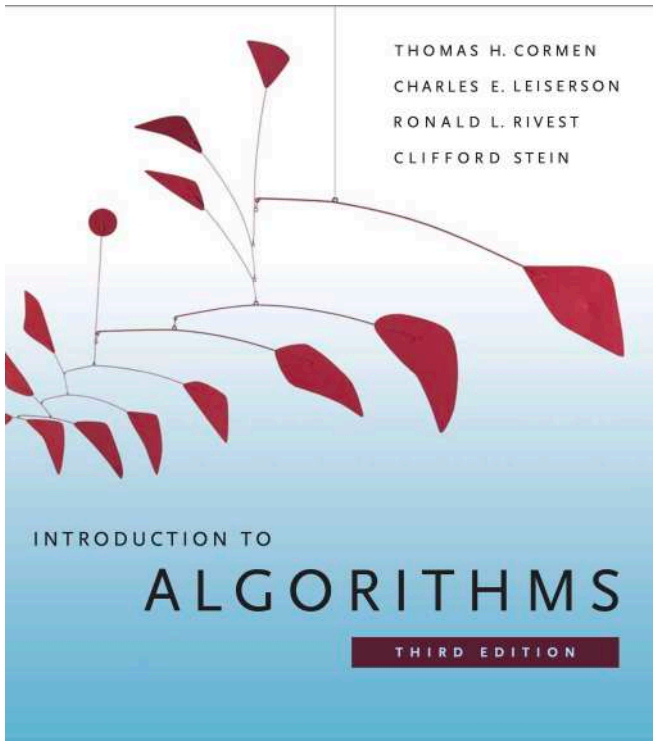
Ablaufplan Vorlesung und Portfolioleistungen



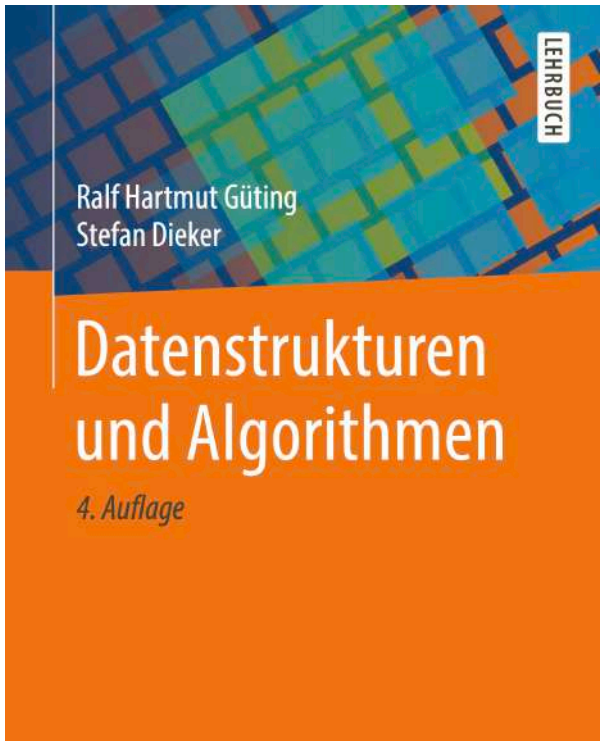
Literaturempfehlungen



- Inhalte des Kurses gut abgedeckt
- Formulierungen in JAVA
- Code auf Deutsch 🙄

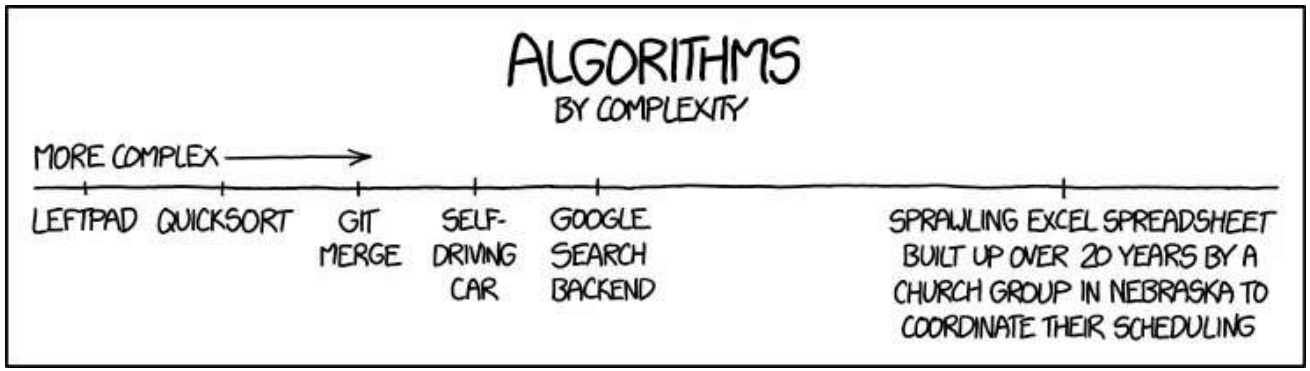


- Inhalte des Kurses gut abgedeckt
- Formulierungen in Pseudocode
- Indizierung beginnend mit 1 🙄
- Fokus auf Theorie



- Inhalte des Kurses gut abgedeckt
- Kein JAVA
- Andere Gliederung

Algorithmen und deren Eigenschaften



Gruppenarbeit (5 min): Definieren Sie den Begriff Algorithmus und geben Sie ein einfaches Beispiel. Was versteht man unter Komplexität? Welche weiteren wichtigen Eigenschaften gibt es?

Algorithmus (informell): Unter einem Algorithmus versteht man eine genau definierte Handlungsvorschift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten.

Algorithmen und deren Eigenschaften

Algorithmus (informell): Unter einem Algorithmus versteht man eine genau definierte Handlungsvorschift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten.

Algorithmus (Eigenschaften):
- Korrektheit
- Komplexität
- Determinusmus
- Vollständigkeit
- Endlichkeit
- Bestimmtheit

Algorithmus (Beispiele):
- Kochrezept
- Bedienungsanleitung
- Notenblatt
- Programmablaufplan

Was genau ist ein Problem?
Was ist eine Art von Problem?
Was ist Lösung eines Problems?

Was ist ein Problem

Beispiel:

Bestimme den größten gemeinsamen Teiler von 144 und 729 👈 konkret

Bestimme den größten gemeinsamen Teiler zweier Zahlen *x* und *y* 👈 allgemein

Also nicht verwirren lassen durch die beiden Ebenen. Saubere Schreibweise und Sprachgebrauch:

Problem: Größter gemeinsamer Teiler

Gegeben: Zwei positive ganze Zahlen *x* und *y*

Gesucht: Die größte ganze Zahl *z* mit der Eigenschaft, dass *z* sowohl *x* als auch *y* teilt.

Eine derartig allgemein beschriebene Aufgabenstellung bezeichnen wir als **Problem**. Eine Aufgabenstellung mit konkreten Ausprägungen von *x* und *y* heißt **Instanz** eines Problems.

Vom Problem zum Algorithmus

- Die Lösung eines Problems besteht nun in einem systematischen Verfahren, das für jede beliebige Instanz eine Lösung liefert
- Das Verfahren ist der Algorithmus zur Lösung des Problems
- Ziele der Veranstaltung
  - Kennenlernen von Standard-Algorithmen für bekannte Problemklassen
  - Analyse deren Eigenschaften (Komplexität, Korrektheit)
  - Datenstrukturen zur Organisation und Strukturierung von Objekten

Beispiel: Collatz-Folge

$$\mathrm{Col}(n) = \begin{cases} \frac{n}{2} & \text{wenn } n \text{ gerade ist,} \\ 3n+1 & \text{wenn } n \text{ ungerade ist.} \end{cases}$$

Hochschule Fulda
University of Applied Sciences

Inhalt

Gliederung

1. **Laufzeit und Komplexität**
2. Sortieren
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

Laufzeitanalyse (1)

- **Idee 1:** Direktes Messen der Laufzeit
  - Wieso ist das keine gute Idee?
  - Viele Faktoren können die Laufzeit eines Programmes beeinflussen
  - Hardware, Betriebssystem, Compilerkonfiguration usw.
  - Nicht aussagekräftig für unterschiedliche Hardware!
  - Dennoch auch in der Wissenschaft sehr verbreitet

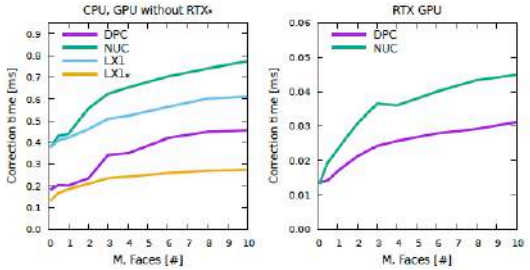


Fig. 10: Run time evaluation of a single MICP-L iteration in synthetic maps of varying sizes on different architectures for CPUs and GPUs without (left) and with RTX capabilities (right).

TABLE III: MICP-L partial run times for CPU, non-RTX-GPU (GPU), and RTX-GPU (RTX) devices.

Device	Simulation	Reduction	SVD
CPU	96.400 %	3.586 %	0.014 %
GPU	94.090 %	0.014 %	5.896 %
RTX	80.337 %	0.376 %	19.287 %

Nur weil es gängige Praxis ist, heißt es noch lange nicht, dass es sinnvoll ist!

Laufzeitanalyse (2)

- **Idee 2:** Bestimmung der benötigten Elementaroperationen eines Algorithmus in Abhängigkeit von der Größe *n* der Eingabe
- Das Laufzeitverhalten des Algorithmus wird als Funktion der benötigten Elementarfunktionen dargestellt
- Was sind Elementarfunktionen?
  - Zugriff auf eine Speicherzelle
  - Kopieren / Verschieben eines Objektes
  - Zeigerdereferenzierungen
  - Zuweisungen
  - Vergleiche
  - …
- Die Zeit, die für diese Operationen benötigt wird, muss mit berücksichtigt werden
- Wir gehen davon aus, dass Zugriffszeiten auf Speicher und Vertauschoperationen normalerweise keinen Aufwand erzeugen

Laufzeitanalyse (3)

- Die Charakterisierung von *n* hängt von der Problemstellung ab
  - Suche eines Elementes in einer verketteten Liste: *n* = Anzahl der Elemente in der Liste
  - Matrixmultiplikation: *n* = Dimension der Matrizen
  - Sortierung einer Liste von n Zahlen: *n* = Anzahl der Zahlen
  - Berechnung der *k*-ten Fibonacci-Zahl: *n* = *k*

Laufzeit ≈ Anzahl der benötigten Elementarfunktionen in Abhängigkeit von *n*

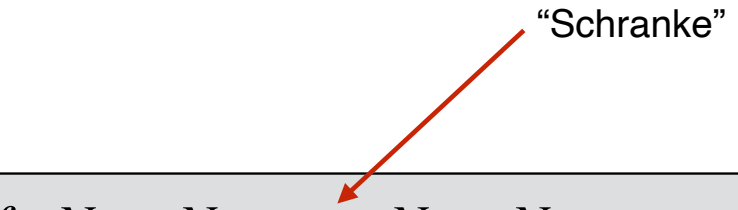
Speicherverbrauch ≈ Anzahl der benötigen Speicherelemente in Abhängigkeit von *n*

Laufzeitanalyse (4)

- Die gemessene Laufzeit ist abhängig von der eingesetzten Hardware und der gewählten Problemrepräsentationen
  - Integer- oder Realzahlrepräsentation?
  - Single oder Double-Precision
  - Branching
  - …
- Dennoch lassen sich durch eine Analyse des Laufzeitverhaltens in Bezug auf n Erkenntnisse über das Strukturelle Verhalten eines Algorithmus bei Variation der Eingabegröße treffen
- Kleine n sind in der Praxis irrelevant
- Interessenten ist das strukturelle Verhalten auf großen Datenmengen
- “Wie ist die zu erwartende Entwicklung der Laufzeit auf der gewählten Hardware wenn ich die Problemgröße verdoppele?”

$\mathcal{O}$-Notation (1)

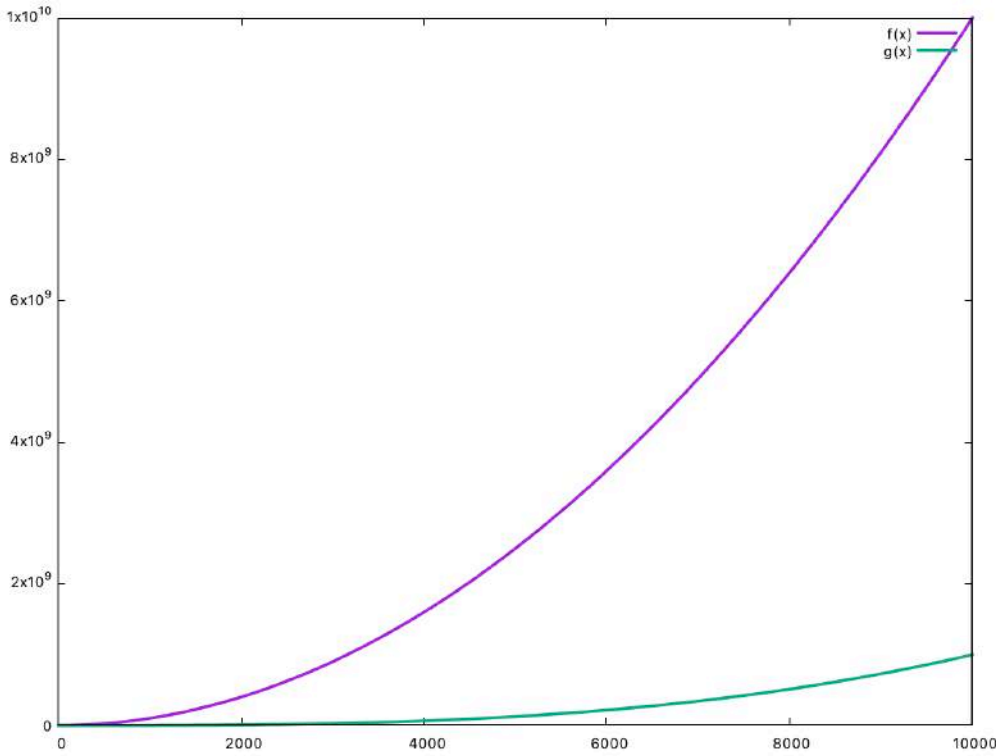
- Die $\mathcal{O}$-Notation charakterisiert die asymptotische Komplexität bzgl. Laufzeit oder Speicherbedarf eines Algorithmus
- Definition $\mathcal{O}$-Notation (formal):

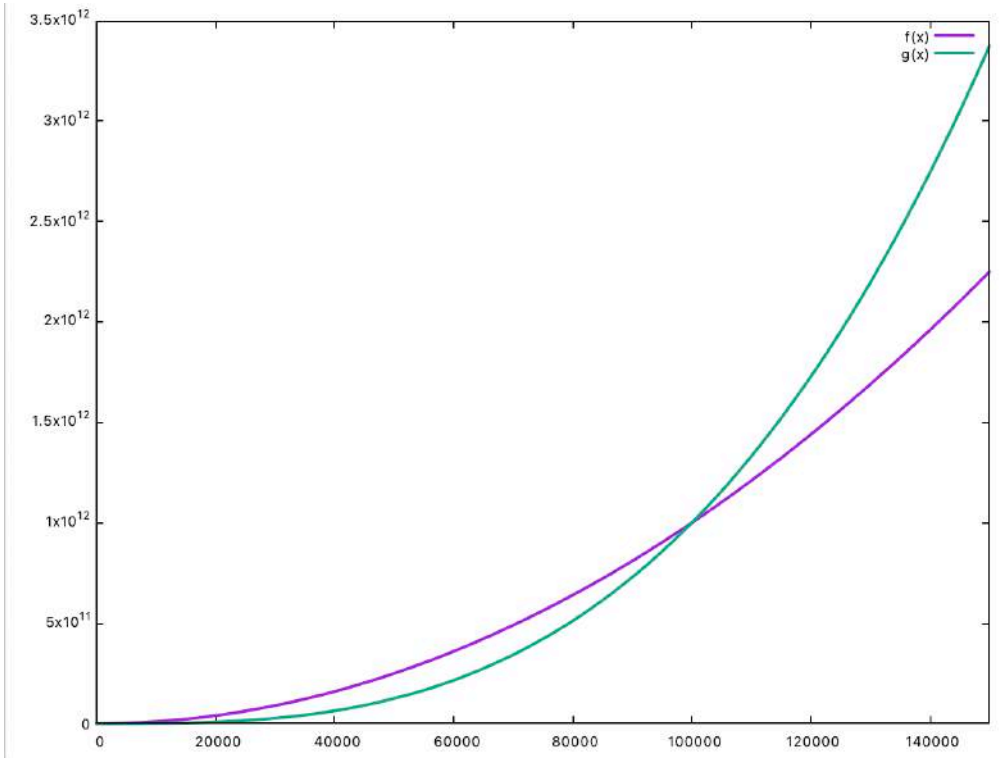


Seien $f : N \to N$ und $s : N \to N$ zwei Funktionen. Die Funktion f ist von der Größenordnung $\mathcal{O}(s)$, wenn es $k \in N$ und $m \in N$ gibt, so dass gilt: Für alle $n \in N$ mit $n \geq m$ ist $f(n) \leq k \cdot s(n)$.

Beispiel

- Vergleiche die beiden Funktionen $f(n) = 100 \cdot n^2$ und $g(n) = 0.001 \cdot n^3$





$\mathcal{O}$-Notation (2)

- k ist unabhängig von n
- $\mathcal{O}(s)$ ist die Menge aller Funktionen, die bezüglich s die Eigenschaft aus der Definition haben
- Bestimmung des dominierenden Terms, ergibt in der Regel die Komplexitätsklasse
- Eliminierung von Konstanten
  - $2 \cdot n \in \mathcal{O}(n)$
  - $n/2 + 1 \in \mathcal{O}(n)$
- Welche Komplexitätsklasse hat $2n^3 + 0.001 \cdot n^2 + 12$?
- $\mathcal{O}(n^3)$
- Bei der Bewertung zählt nur das Maß der Komplexität! Die Faktoren spielen keine Rolle.

Seien $f : N \to N$ und $s : N \to N$ zwei Funktionen. Die Funktion f ist von der Größenordnung $\mathcal{O}(s)$, wenn es $k \in N$ und $m \in N$ gibt, so dass gilt: Für alle $n \in N$ mit $n \geq m$ ist $f(n) \leq k \cdot s(n)$.

$\mathcal{O}$(n)-Notation (3)

- Wichtige Komplexitätsklassen:

$\mathcal{O}$(1)
konstante Komplexität, die Laufzeit hängt nicht von der Datenmenge ab.
z.B. Arrayzugriff, Hashtable

$\mathcal{O}$(n)
lineare Komplexität, die Laufzeit ist propertional zur Datenmenge.
z.B. Schleife über ein Array um einen Wert zu finden, Einlesen einer Treffermenge aus der Datenbank

$\mathcal{O}(n^2)$
quadratische Komplexität, eine doppelte Datenmenge vervierfacht die Laufzeit
z.B. Bubble-Sort

$\mathcal{O}$(log n)
logarithmische Komplexität, wird die Datenmenge jeweils verdoppelt, steigt die Laufzeit linear an
z.B. Suchbäume

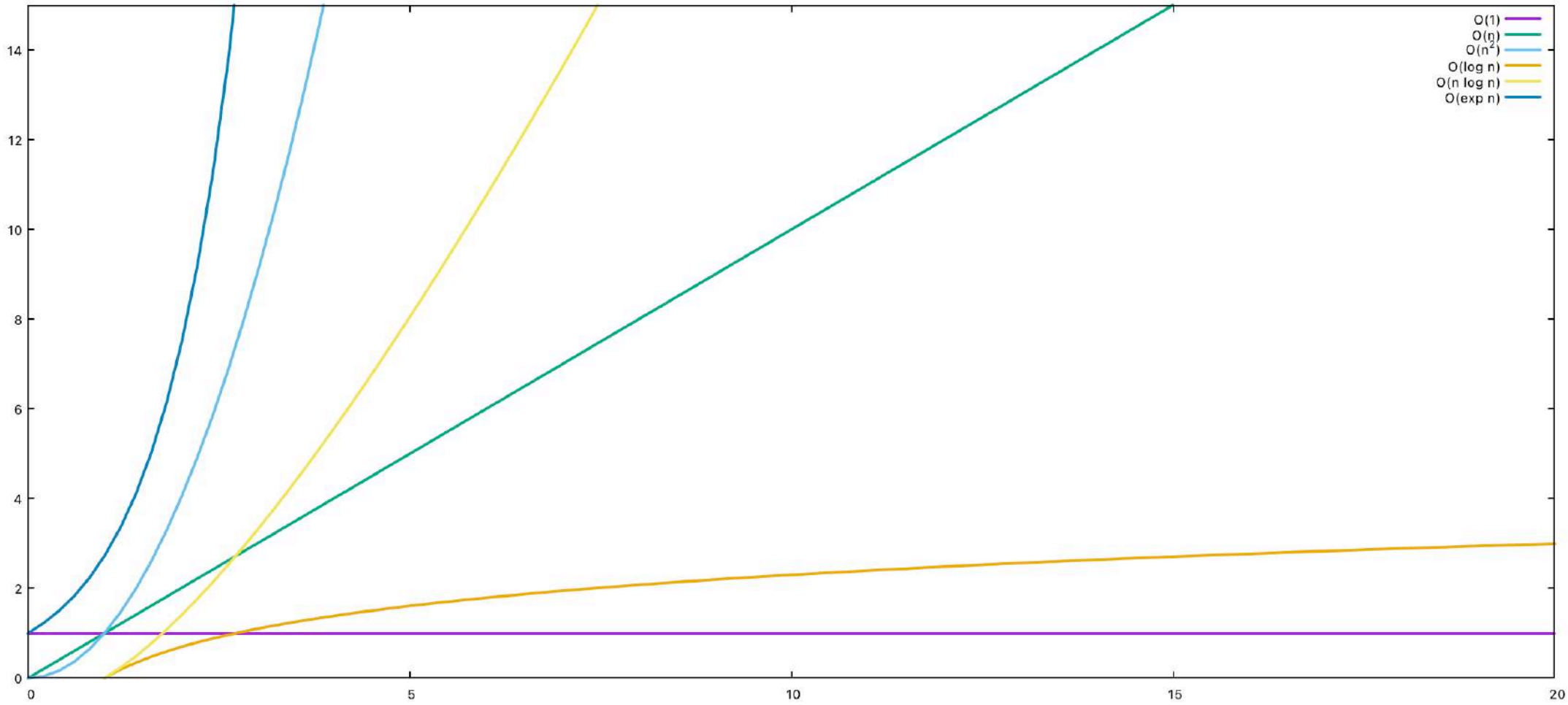
$\mathcal{O}$(n log n)
superlineare Komplexität, liegt zwischen $\mathcal{O}(n)$ und $\mathcal{O}(n^2)$. Tritt zum Beispiel auf, wenn eine Schleife über eine Baumsuche gebildet wird.
z.B. optimierte Sortieralgorithmen wie Quicksort

$\mathcal{O}(2^n)$
exponentielle Komplexität, die Laufzeit verdoppelt sich, wenn die Datenmenge um eine Einheit größer wird.
z.B. Bilden aller Paare einer Menge, Türme von Hanoi als rekursiver Algorithmus

$\mathcal{O}$(n!)
faktorielle Komplexität, die Laufzeit wächst mit der Fakultät der Datenmenge.
z.B. Problem des Handlungsreisenden

https://wiki.selfhtml.org/wiki/Laufzeitkomplexität

$\mathcal{O}$-Notation (4)



Best, Average und Worst-Case

- Üblicherweise leicht zu bestimmen sind Best-Case und Worst-Case-Szenarien weil konstruierbar
- Beispiel: Lineares Suchen eines Elements in einer sortierten Liste
  - Best-Case?
  - Worst-Case?
  - Average-Case?
- Average-Cast ist in der Praxis die relevanteste Größe
- In der Regel aber nur schwer abzuschätzen
- Worst-Case-Abschätzungen sind oft inadäquat, wenn diese Fälle selten vorkommen
- Ebenso Best-Case
- Kommen Worst-Case-Szenarien praktisch nie vor, kann auch ein Algorithmus mit schlechtem Worst-Case praktisch gut sein!

Beispiel: Minimum

```
// Given: int array a with n (n >=2) elements.
// Find the minimum value

int min = a[0];
for(int i = 1; i < n; i++)
    if(a[i] < min) min = a[i];
```

Worst Case? $\rightarrow \mathcal{O}(n)$

Best Case? $\rightarrow \mathcal{O}(n)$

Average Case? $\rightarrow \mathcal{O}(n)$

Beispiel: Suche nach Duplikaten

```
// Given: array a with n elements. Test if
// two elements in the array are indentical
```

```
for(int i = 0; i < n - 1 && !found; i++)
```

Was ändert sich?

```
boolean found = false;
for(int i = 0; i < n - 1; i++)
  for(int j = i + 1; j < n; j++)
      if(a[i] == a[j]) found = true;
```

Wie viele werden Schritte benötigt

$$n-1+n-2+n-3+\ldots+2+1$$

$$\sum_{i=1}^{n-1} i \quad = \frac{n(n-1)}{2} \quad = \frac{n^2-n}{2} \quad \in \mathcal{O}(n^2)$$

Beispiel Fakultät

- Welche Komplexitätsklasse hat die Berechnung der Fakultät?
- Erinnerung: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$

```
public static int f(int n)
{
  if (n < 0) throw new RuntimException("n has to be >= 0");
  if (n > 12) throw new RuntimeException("n has to be <= 12");
  int result = 1;
  for(int i = 0; i <= n; i++)
    result = result * i;
  return result;
}
```

Beispiel Fakultät - Rekursive Variante

- Erinnerung: $f(n) = n \cdot f(n-1)$

```
public static int f(int n)
{
  if(n == 0) return 1;
  else return n * f(n-1);
}
```

Beispiel: Suche nach ‘0’

```
// Given: character array with n elements
// consisting only of '0's and '1's

for(i = 0; i < n && a[i] != '0'; i++)
```

Worst Case? $\rightarrow \mathcal{O}(n)$

Best Case? $\rightarrow \mathcal{O}(1)$

Average Case?

Erwartungswert = $\sum$ Ereignis * Wahrscheinlichkeit

$$= 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + \cdots = \sum_{i=1}^{n} i \cdot \frac{1}{2^i} < 2$$

$$\rightarrow \mathcal{O}(1)$$