
Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender (Übungen)

sowie viele Tutoren



Allgemeine Lernziele **Vorlesung**

Einem Vortragenden **zuhören zu lernen**,
... der über ein nicht ganz triviales Thema referiert

Dem Vortragenden beim Vortrag **gedanklich folgen** können

Vorbereitung für das selbständige Erarbeiten der Inhalte

Erarbeitung durch

- Nacharbeitung der Präsentationen
- Lösen von Übungsaufgaben
- Diskussion in Übungsgruppe



Teilnehmerkreis und Voraussetzungen

Studiengänge

- Bachelor **Informatik**
- Bachelor **IT-Sicherheit**
- Bachelor **Mathematik in Medizin und Lebenswissenschaften**
- Bachelor **Medieninformatik**
- Bachelor **Medizinische Informatik**
- Bachelor **Medizinische Ingenieurwissenschaft**
- Bachelor **Robotik und Autonome Systeme**

Vorausgesetzte Kenntnisse

- Einführung in die Programmierung
- Übliche Lernfähigkeit und Aufnahmebereitschaft aus dem 2. Fachsemester

Es sind manchmal Studierende aus anderen Studiengängen und höheren Fachsemestern dabei

Organisatorisches: Übungen

- **Start:** Siehe Moodle
- **Übungen:** Verschiedene Gruppen, Anmeldung über Moodle
- **Übungsaufgaben** stehen jeweils kurz nach der Vorlesung am Freitag über Moodle bereit
- Besprechung erfolgt in der Übung, Präsenzaufgaben in der Übung helfen bei der Lösungen der Aufgaben
- Aufgaben sollen in einer **2-er Gruppe** bearbeitet werden
- **Abgabe der Lösungen** erfolgt bis Donnerstag in der jeweils folgenden Woche nach Ausgabe bis 16 Uhr über Moodle

Prüfungsvorleistungen

1. Sie müssen durch Abgabe von Lösungen für Übungsaufgaben 160 von 240 möglichen Punkten erzielen (ca. 67%), um zur Prüfung zugelassen zu werden
 - Bei Programmieraufgaben ist nur Julia als Programmiersprache zugelassen
2. In der Vorlesung werden ggf. Quizfragen gestellt, die Sie beantworten können (über Moodle oder Zettel)
 - Richtige Antworten bringen ihnen Bonuspunkte, mit denen Sie leichter über 160 Punkte kommen können (siehe Punkt 1)
 - Näheres hierzu wird noch bekanntgegeben
3. Mit einer Midterm-Programmieraufgabe können Sie weitere Bonuspunkte erwerben
 - Näheres hierzu wird noch bekanntgegeben

Präsentationen der Vorlesungen

- Auf der Moodle-Seite zum Kurs finden Sie rechtzeitig vor der jeweiligen Vorlesung die von mir für die Vorlesung verwendeten Präsentationen zum Herunterladen
- Da Animationen eine Rolle spielen, laden Sie bitte die jeweilige pptx-Version herunter
 - Pdf dient nur zu groben Orientierung
 - Im Pdf können Überlappungen vorkommen ...
 - ... und Animationen sind nicht sichtbar

Danksagungen in Präsentationen

Kein Dozent erfindet alles selbst neu

- Es wäre dumm, sich nicht von guten Präsentationen inspirieren zu lassen (und was gut ist, beurteilt jemand der ihr gesamtes Studium überblickt)
- Danksagungen und Zitierungen auf Präsentationen gehören zur guten wissenschaftlichen Praxis
- Wenn ein Dozent das nicht macht, ...



Das „Skript“

Die Zuordnung der Kapitel zu der Vorlesung ist leicht

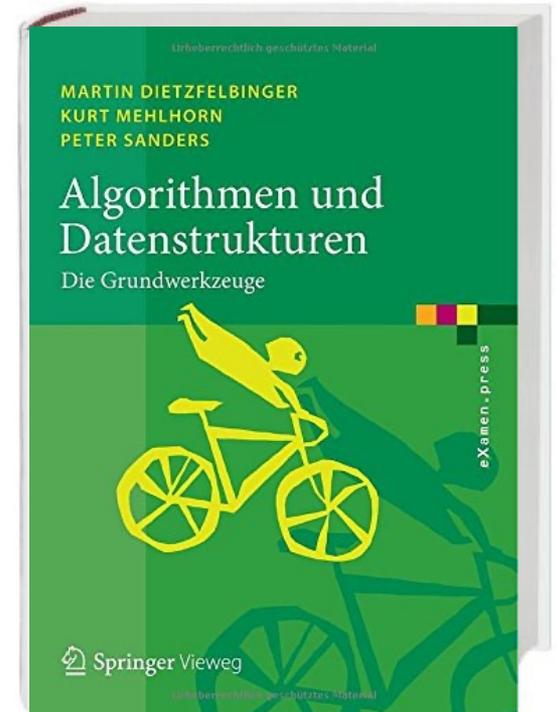
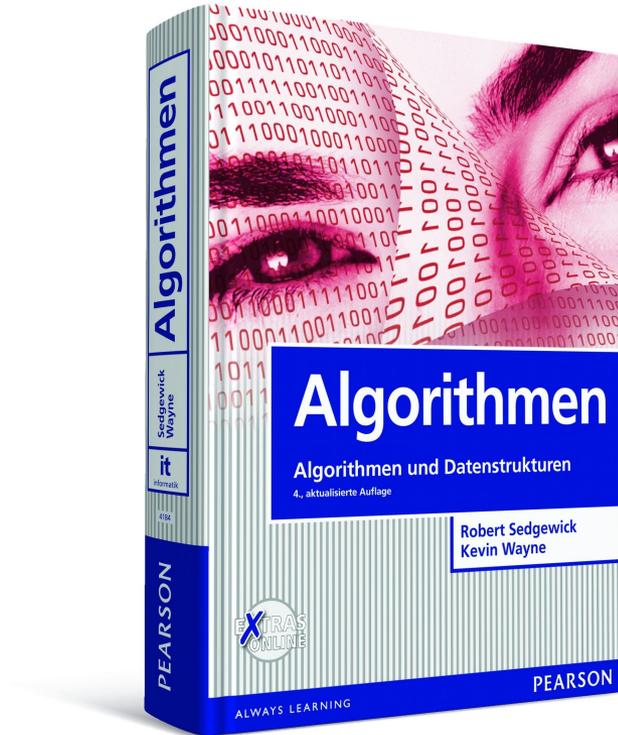
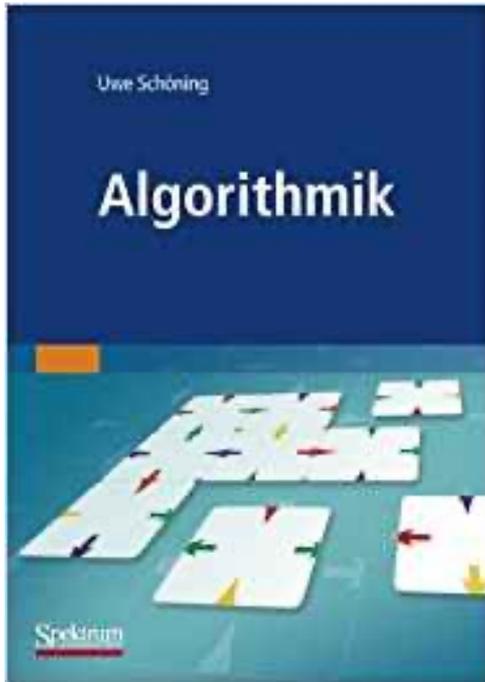


$\frac{3}{4}$ kommt auch dran

Der Rest kommt in den nächsten Semestern dran



Zusätzlich empfohlene Literatur



Spezielle Lernziele in diesem Kurs

- Weg **vom Problem zum Algorithmus** gehen können
 - **Auswahl** eines Algorithmus aus Alternativen unter Bezugnahme auf vorliegende Daten und deren Struktur
 - **Entwicklung** eines Algorithmus mitsamt geeigneter Datenstrukturen (Terminierung, Korrektheit, ...)
- **Analyse von Algorithmen** durchführen
 - Anwachsen der Laufzeit bei Vergrößerung der Eingabe
- Erste Schritte in Bezug auf die **Analyse von Problemen** gehen können
 - Ja, Probleme sind etwas anderes als Algorithmen!
 - Probleme können in gewisser Weise „schwer“ sein
 - Prüfung, ob Algorithmus optimal

Beispielproblem: Summe der Elemente eines Feldes $A[1..n]$ bestimmen

• Algorithmus?

- $\text{summe}(A) = \sum_{i=1}^n A[i]$

- Programm: $i=1$

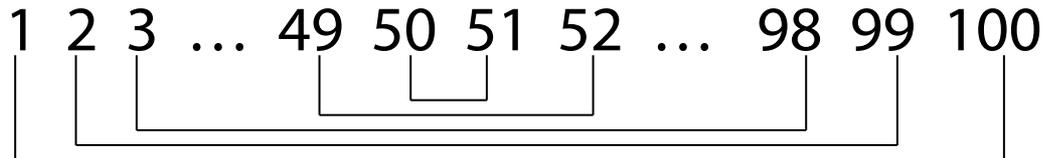
```
function summe(A)
    s = 0
    for i = 1:length(A)
        s = s + A[i]
    end
    return s
end
```

- Aufwand?
- Wenn A n Elemente hat, n Schritte!
- Der Aufwand wird *linear* genannt

Spezialisierung des Problems

- Vorwissen: $A[i] = i$
- Das Problem wird sehr viel einfacher!

Ausnutzen der Einschränkung



Summe jedes Paares: 101

50 Paare: $101 * 50 = 5050$

- Programm:

```
function summe (A)
    n = length (A)
    return (n+1) * (n/2)
end
```

- Nach Carl Friedrich Gauß (ca. 1786)
- Aufwand?
- Konstant, d.h. hängt (idealisiert!) nicht von n ab
- Entwurfsmuster: Ein-Schritt-Berechnung (konstanter Aufwand)

Das erste Problem: Summe der Elemente

- **Gegeben: $A[1..n] : \mathbb{N}$**
 - Feld (Array) A von n Zahlen aus \mathbb{N} (natürliche Zahlen)
- **Gesucht:**
 - Transformation \mathbf{S} auf A , so dass gilt:
 - $s = \sum_{i \in \{1, \dots, n\}} A[i]$
 - Also: Gesucht ist ein Verfahren \mathbf{S} , so dass $\{\mathbf{P}\} \mathbf{S} \{\mathbf{Q}\}$ gilt (Notation nach [Hoare](#))
 - Vorbedingung \mathbf{P} : **true** (keine Einschränkung)
 - Nachbedingung \mathbf{Q} : $s = \sum_{i \in \{1, \dots, n\}} A[i]$

Das zweite Problem: Summe der Elemente

- **Gegeben: $A[1..n] : \mathbb{N}$**
 - Feld (Array) A von n Zahlen aus \mathbb{N} (natürliche Zahlen)
- **Gesucht:**
 - Transformation S von A , so dass gilt:
 - $s = \sum_{i \in \{1, \dots, n\}} A[i]$
 - Also: Gesucht ist ein Verfahren S , so dass $\{P\} S \{Q\}$ gilt (Notation nach Hoare)
 - Vorbedingung P : $\forall i \in \{1 \dots n\}: A[i] = i$
 - Nachbedingung Q : $s = \sum_{i \in \{1, \dots, n\}} A[i]$

Programmiersprache julia

- Programmcode auf den Präsentationen ist in Julia geschrieben
 - Sollte intuitiv verständlich sein
- Algorithmen von den Präsentationen können ausgeführt werden
 - Funktionsweise gut nachzuvollziehen durch schrittweises Ausführen und Debuggen
 - Zip-Datei mit Programmcode von den Folien im Moodle
- Einführung in Julia heute als Teil der Vorlesung

Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender (Übungen)

sowie viele Tutoren



Ein neues Problem: In-situ-Sortierproblem

- **Gegeben: $A[1..n] : \mathbb{N}$**
 - Feld (Array) A von n Zahlen aus \mathbb{N} (natürliche Zahlen)
- **Gesucht:**
 - Transformation S von A , so dass gilt: $\forall 1 \leq i < j \leq n: A[i] \leq A[j]$
 - Nebenbedingung: Es wird intern kein weiteres Feld gleicher (oder auch nur fast gleicher Größe) verwendet
 - Also: Gesucht ist ein Verfahren S , so dass $\{P\} S \{Q\}$ gilt (Notation nach Hoare)
 - Vorbedingung P : **true** (keine Einschränkung)
 - Nachbedingung Q : $\forall 1 \leq i < j \leq n: A[i] \leq A[j]$
 - Nebenbedingung: nur „konstant“ viel zusätzlicher Speicher (feste Anzahl von Hilfsvariablen), nur Vergleiche erlaubt

In-situ-Sortieren: Problemanalyse

- Felder erlauben **wahlfreien** Zugriff auf Elemente
 - **Zugriffszeit** für ein Feld **konstant** (d.h. sie hängt nicht vom Indexwert ab)
 - Idealisierende Annahme (gilt nicht für moderne Computer)
- Es gibt keine Aussage darüber, ob die Feldinhalte schon sortiert sind, eine willkürliche Reihenfolge haben, oder umgekehrt sortiert sind
 - Vielleicht lassen sich solche „**erwarteten Eingaben**“ aber in der Praxis feststellen
- **Aufwand das Problem zu lösen:** Man kann leicht sehen, dass jedes Element „falsch positioniert“ sein kann
 - **Mindestaufwand** im allgemeinen Fall: n Bewegungen
 - **Maximalaufwand** in Abhängigkeit von n ?

Aufwand zur Lösung eines Problems

- Gegeben ein **Problem** (hier: In-situ-Sortierproblem)
 - Damit verbundene Fragen:
 - Wie „langsam“ muss ein Algorithmus sein, damit alle möglichen Probleminstanzen korrekt gelöst werden?
 - Oder: Wenn wir schon einen Algorithmus haben, können wir noch einen „substantiell besseren“ finden? Was müssen wir investieren?
- Jede Eingabe **A** stellt eine **Probleminstanz** dar
- Notwendiger Aufwand in Abhängigkeit von der Größe der Eingabe heißt **Komplexität eines Problems**
 - Anzahl der notwendigen Verarbeitungsschritte in Abhängigkeit der Größe der Eingabe (hier: Anzahl der Elemente des Feldes **A**)
 - Komplexität durch jeweils „schlimmste“ Probleminstanz bestimmt
 - Einzelne Probleminstanzen können evtl. weniger Schritte benötigen

Entwurfsmuster / Entwurfsverfahren

- Schrittweise Berechnung
 - Beispiel: Bestimmung der Summe eines Feldes durch Aufsummierung von Feldelementen
- Ein-Schritt-Berechnung
 - Beispiel: Bestimmung der Summe eines Feldes ohne die Feldelemente selbst zu betrachten (geht nur unter Annahmen)
- Verkleinerungsprinzip
 - Beispiel: Sortierung eines Feldes
 - Unsortierter Teil wird immer kleiner, letztlich leer
 - Umgekehrt: Sortierter Teil wird immer größer, umfasst am Ende alles → Sortierung erreicht

Laufzeitanalyse

- Laufzeit als **Funktion der Eingabegröße**
- Verbrauch an Ressourcen: **Zeit, Speicher**, Bandbreite, Prozessoranzahl, ...
- Laufzeit bezogen auf serielle Maschinen mit wahlfreiem Speicherzugriff
 - **von Neumann-Architektur ...**
 - ... und Speicherzugriffszeit als konstant angenommen
- Laufzeit kann von der Art der Eingabe abhängen (**besten Fall, typischer Fall, schlechtesten Fall**)
- Meistens: schlechtesten Fall betrachtet

Aufwand für Zuweisung, Berechnung, Vergleich?

	Zeit	Wie oft?
<code>function insertion_sort(A)</code>		
1: <code>for j = 2:length(A)</code>	c_1	n
2: <code>key = A[j]</code>	c_2	$n - 1$
3: <code># insert A[j] in A[1..j - 1]</code>		
4: <code>i = j - 1</code>	c_4	$n - 1$
5: <code>while i > 0 && A[i] > key</code>	c_5	$\sum_{j=2}^n t_j$
6: <code>A[i + 1] = A[i]</code>	c_6	$\sum_{j=2}^n (t_j - 1)$
7: <code>i = i - 1</code>	c_7	$\sum_{j=2}^n (t_j - 1)$
<code>end</code>	c_7	$\sum_{j=2}^n (t_j - 1)$
8: <code>A[i + 1] = key</code>	c_8	$n - 1$
<code>end</code>		
<code>end</code>		

t_j = Anzahl der Durchläufe der *while*-Schleife im j -ten Durchgang.

$c_1 - c_8$ = un spezifizierte Konstanten.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

Bester Fall: Feld ist aufsteigend sortiert

Ist das Array bereits aufsteigend sortiert, so wird die *while*-Schleife jeweils nur einmal durchlaufen: $t_j = 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Also ist $T(n)$ eine **lineare Funktion** der Eingabegröße n .

Schlechtester Fall: Feld ist absteigend sortiert

Ist das Array bereits absteigend sortiert, so wird die *while*-Schleife maximal oft durchlaufen: $t_j = j$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

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

$$\begin{aligned} \sum_{j=2}^n (j - 1) &= \frac{n(n-1)}{2} \\ &= \sum_{j=2}^n j - \sum_{j=2}^n 1 = (n(n+1)/2 - 1) - (n-1) \\ &= n(n+1)/2 - n = n^2/2 + n/2 - n \\ &= n^2/2 - n/2 = n(n-1)/2 \end{aligned}$$

Schlechtester Fall: Feld ist absteigend sortiert

Ist das Array bereits absteigend sortiert, so wird die *while*-Schleife maximal oft durchlaufen: $t_j = j$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

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

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \frac{n(n-1)}{2}$$

Also ist $T(n)$ eine **quadratische Funktion** der Eingabegröße n .

$$T(n) = c_0 n^2 + \dots$$

Schlimmster vs. typischer Fall

Meistens geht man bei der Analyse von Algorithmen vom (*worst case*) aus.

- *worst case* Analyse liefert **obere Schranken**
- In vielen Fällen ist der *worst case* die Regel
- Der aussagekräftigere (gewichtete) Mittelwert der Laufzeit über alle Eingaben einer festen Länge (*average case*) ist oft bis auf eine multiplikative Konstante nicht besser als der *worst case*.
- Belastbare Annahmen über die mittlere Verteilung von Eingaben sind oft nicht verfügbar.

Eine andere "Idee" für Sortieren

- Gegeben: $a = [4, 7, 3, 5, 9, 1]$
- Gesucht: In-situ-Sortierverfahren

```
function selection_sort(A)
    for i = 1:length(A)
        min = i
        for j = (i + 1):length(A)
            if A[j] < A[min] # find the minimum in the unsorted part
                min = j
            end
        end
        x = A[i]           # swap the found minimum into sorted part
        A[i] = A[min]
        A[min] = x
    end
end
```

The diagram illustrates the state of an array A during the selection sort process. The array is represented as a horizontal bar divided into three sections. The first section, from index 1 to $i-1$, is shaded green and labeled "sortiert". The second section, at index i , contains the element x . The third section, from index $i+1$ to n , is labeled "????". Arrows point from the labels "1", "i", and "n" below the array to their respective indices. An arrow labeled "A" points to the start of the array.

Sortieren durch Auswählen (Selection-Sort)

- Gleiches **Entwurfsmuster: Verkleinerungsprinzip**
- Aufwand im **schlechtesten Fall?**

- $T(n) = c_1 n^2 + \dots$

- Aufwand im **besten Fall?**

- $T(n) = c_2 n^2 + \dots$



- Sortieren durch Auswählen scheint also noch schlechter zu sein als Sortieren durch Einfügen !
- Kann sich jemand eine Situation vorstellen, in der man trotzdem zu Sortieren durch Auswählen greift?
 - Was passiert, wenn die Elemente sehr groß sind?
 - Verschiebungen sind aufwendig

Vergleich der beiden Algorithmen

- Anzahl Vergleiche?
- Anzahl Zuweisungen?
- Berechnungen?

- Was passiert, wenn die Eingabe immer weiter wächst?
 - $n \mapsto \infty$
- Asymptotische Komplexität eines Algorithmus
- Die Konstanten c_i sind nicht dominierend
- Lohnt es sich, die Konstanten zu bestimmen?
- Sind die beiden Algorithmen substantiell verschieden?

Sorting



```
function insertion_sort(A)
  for j = 2:length(A)
    key = A[j]
    # insert A[j] into the sorted sequence A[1..j - 1]
    i = j - 1
    while i > 0 && A[i] > key
      A[i + 1] = A[i]
      i = i - 1
    end
    A[i + 1] = key
  end
end
```

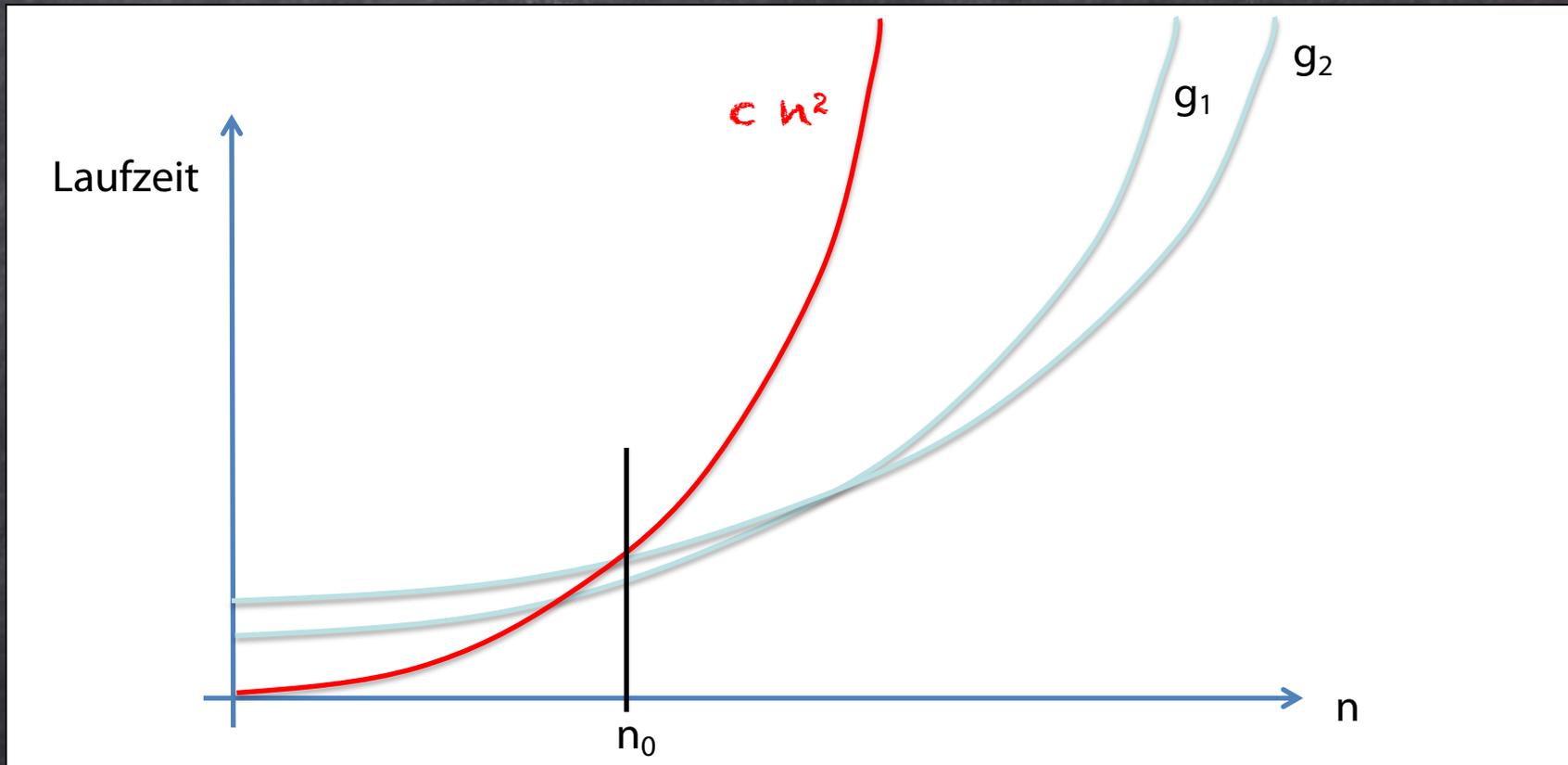
```
function selection_sort(A)
  for i = 1:length(A)
    min = i
    for j = (i + 1):length(A)
      if A[j] < A[min] # find the minimum in the unsorted part
        min = j
      end
    end
    x = A[i]           # swap the found minimum into sorted part
    A[i] = A[min]
    A[min] = x
  end
end
```

Algorithmen vergleichbar?

Ja, bezüglich der
asymptotischen Komplexität
der Algorithmen

Quadratischer Aufwand

- Algorithmus 1: $g_1(n) = b_1 + c_1 * n^2$
- Algorithmus 2: $g_2(n) = b_2 + c_2 * n^2$



Oberer „Deckel“: O-Notation

- Charakterisierung von Algorithmen durch Menge von Funktionen

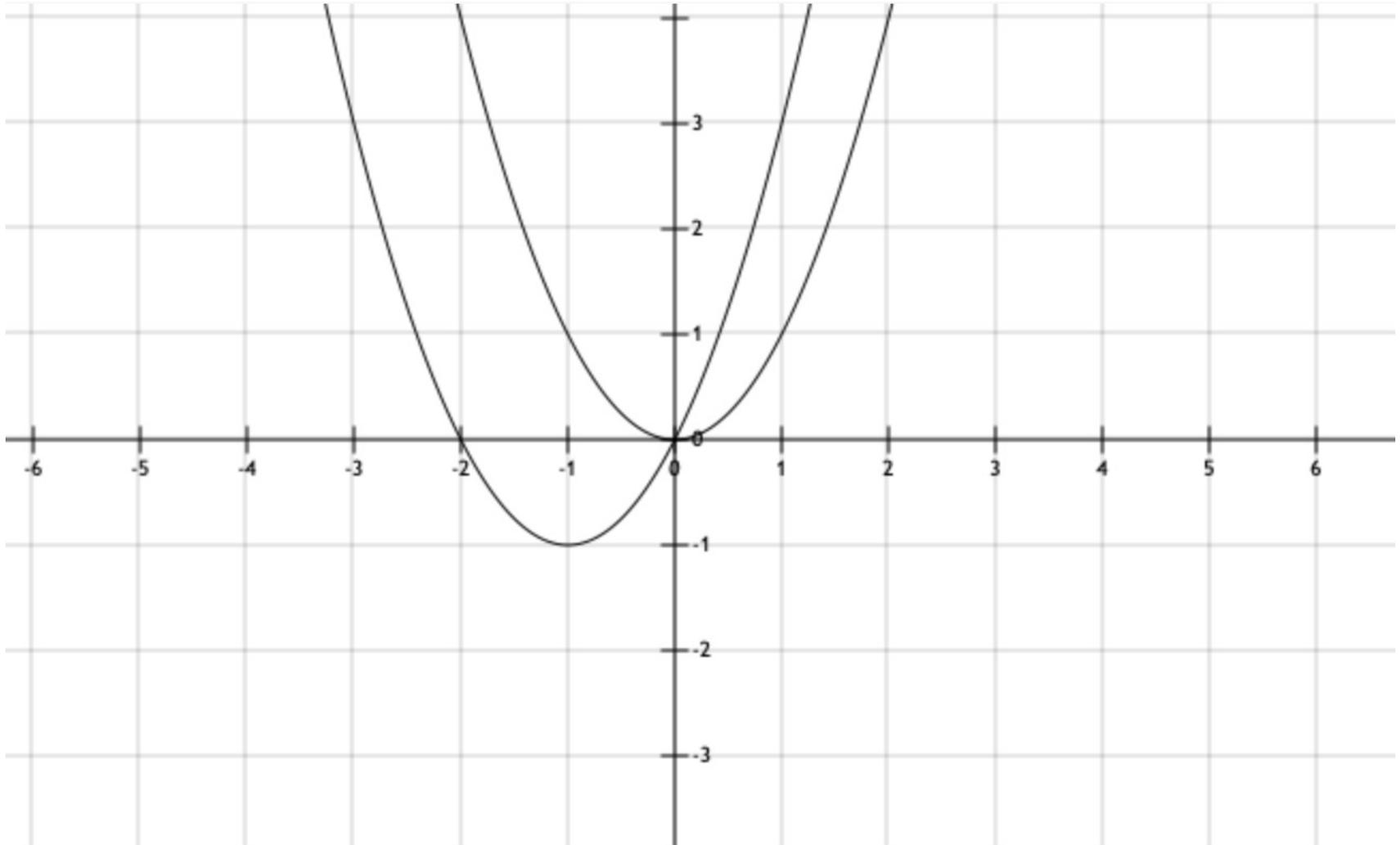
$$O(f(n)) = \{ g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c > 0, \text{ so dass } \forall n \geq n_0: g(n) \leq c \cdot f(n) \}$$

- Sei $f(n) = n^2$
- $g_1 \in O(n^2)$
- $g_2 \in O(n^2)$
- g_1 und g_2 sind „von der gleichen Art“
- $O(f(n))$ definiert „Klasse“ von Funktionen

Erstmals vom deutschen Zahlentheoretiker Paul Bachmann in der **1894** erschienenen zweiten Auflage seines Buchs Analytische Zahlentheorie verwendet. Verwendet auch vom deutschen Zahlentheoretiker Edmund Landauer, daher auch Landau-Notation genannt (**1909**) [Wikipedia 2015]

In der Informatik populär gemacht durch Donald Knuth, In: The Art of Computer Programming: Fundamental Algorithms, Addison-Wesley, **1968**

O-Notation: $O(n^2) = O(n^2 + 2n)$

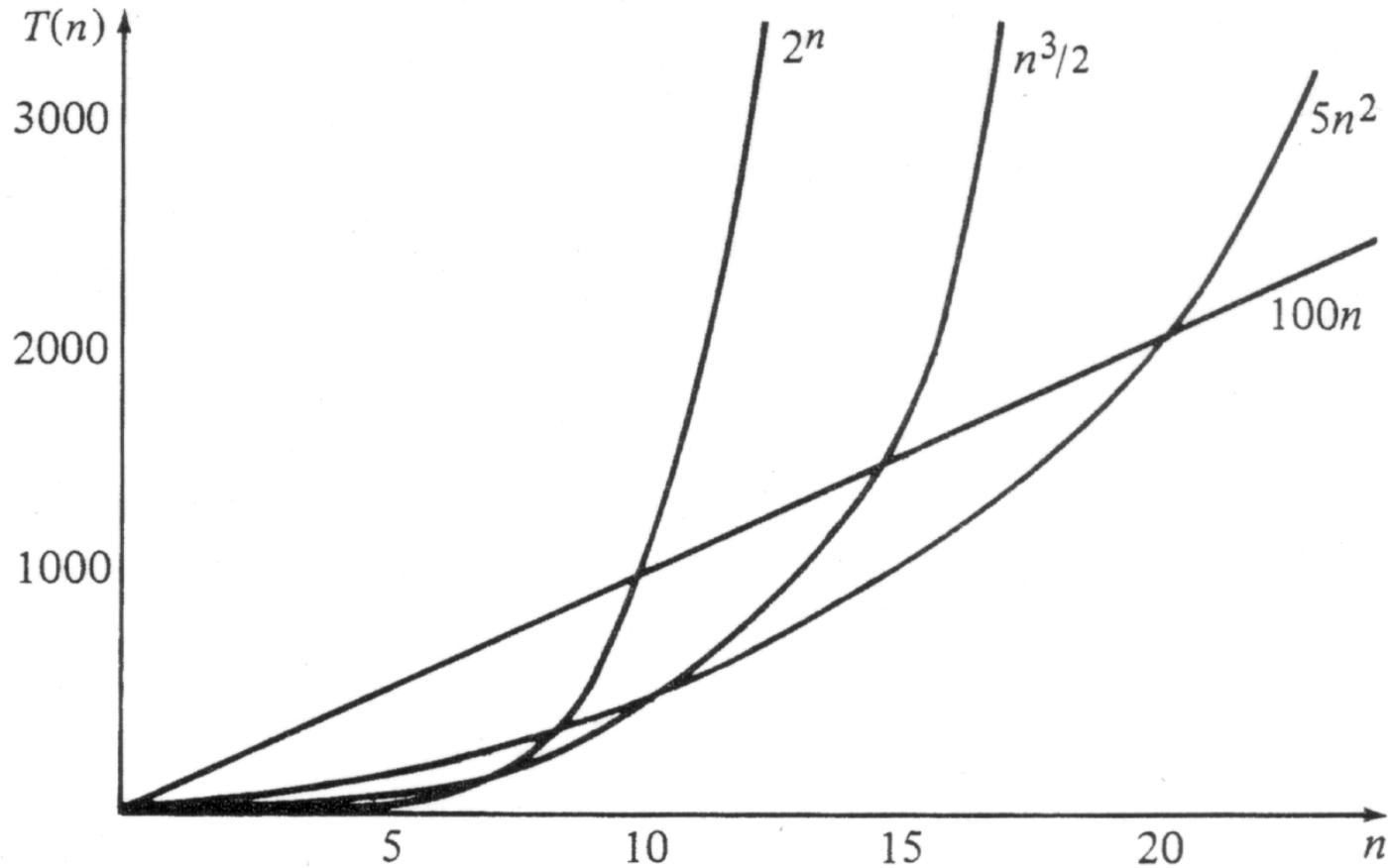


O-Notation

$$O(f(n)) = \{ g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c > 0, \text{ so dass } \forall n \geq n_0: g(n) \leq c \cdot f(n) \}$$

- Quadratischer Aufwand: $O(n^2)$
- Linearer Aufwand: $O(n)$
- Konstanter Aufwand: $O(1)$

Laufzeiten



Zusammenfassung: Entwurfsmuster

- In dieser Vorlesungseinheit:
 - Schrittweise Berechnung
 - Ein-Schritt-Berechnung
 - Verkleinerungsprinzip
- Analyse der Laufzeit: O-Notation
- Nächste Vorlesung
 - Entwurfsmuster Teile und Herrsche