Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

2. Algorithmische Grundlagen/Eigenschaften

Informationstechnik II

Wir zeichnen auf

Prof. Dr.-Ing. Eric Sax

Inhalt IT2

1. Einführung
- Aufbau Modul IT 2 / Verbindung zu IT1 und anderen Lehrveranstaltungen
- Anwendungsbeispiele und Motivation
- Definitionen und Begriffe

Algorithmische Grundlagen/Eigenschaften
- Definition Algorithmus et al.
- Klassifikation von Algorithmen (Komplexität, Problemstellung, Anwendung)
- Merkmale
- Algorithmen-Analyse (Laufzeit, Effizienz, Effektivität)

3. Sortieralgorithmen
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Definition: Algorithmus

- Wer war Musa al-Chwarizimi? (أبو جعفر محمد بن موسى الخوارزمي)
  - Auf ihn geht der Begriff Algorithmus zurück.
  - Er führte Ziffer "null" aus dem indischen in das arabische Zahlensystem ein.
- Algorithmen sind …
  - … genau definierte Handlungsvorschriften zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten
- Beispiele:
  - Kochrezept
  - Gebrauchsanleitungen
  - Hilfen zum Ausfüllen von Formularen
  - Waschmaschinenprogramme

Denkmal des Abu Dscha'far Muhammad ibn Musa al-Chwarizmi

Definition: Algorithmus

- Algorithmus im engeren Sinne der Informatik
  - Gegenstand einiger Spezialgebiete der Theoretischen Informatik (Algorithmentheorie, Komplexitätstheorie, Berechenbarkeitstheorie)
  - Als Computerprogramme und elektronische Schaltkreise steuern sie in Form von konkreten Funktionen Computer und andere Maschinen.

<zur Erklärung der vorangegangenen Folie>

- Ein Algorithmus ist eine wohldefinierte Rechenvorschrift, die eine Größe oder eine Menge von Größen als Eingabe verwendet und eine Größe oder eine Menge von Größen als Ausgabe erzeugt.
- Es ist ein Hilfsmittel, um ein genau festgelegtes Rechenproblem zu lösen. Die Formulierung des Problems legt in allgemeiner Form die benötigte Eingabe-Ausgabe-Beziehung fest. Der Algorithmus beschreibt eine spezifische Rechenvorschrift zum Erhalt dieser Beziehung.

Formale Definition

- Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn folgende Eigenschaften gelten:
  - <u>Finitheit</u>: das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein
  - <u>Ausführbarkeit</u>: jeder Schritt des Verfahrens muss tatsächlich ausführbar sein
  - <u>Dynamische Finitheit</u>: das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen (Platzkomplexität)
  - <u>Terminierung</u>: das Verfahren darf nur endlich viele Schritte benötigen (Zeitkomplexität)

Praktische Eigenschaften

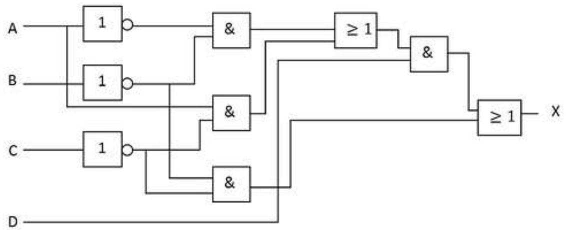
- Der Begriff Algorithmus ist in praktischen Bereichen oft auf die folgenden Eigenschaften eingeschränkt:
  - Determiniertheit: der Algorithmus muss bei den selben Voraussetzungen das gleiche Ergebnis liefern
  - Determinismus: die nächste anzuwendende Regel im Verfahren ist zu jedem Zeitpunkt eindeutig definiert

V1 Ende

Beschreibung von Algorithmen

Problem-stellung → Lösung → Algorithmus → Beschrei-bung → Diskussion

- Lösung eines Problems:
  - Finden einer algorithmischen Beschreibung des Lösungswegs
- Beschreibung mittels:
  - der natürlichen Sprache
  - als Computerprogramm
  - als Hardwareentwurf
  - Graphische Beschreibungssprache (z.B. UML)
  - mit Pseudo Code
  - grafisch, z.B: Nassi Shneiderman Diagrammen

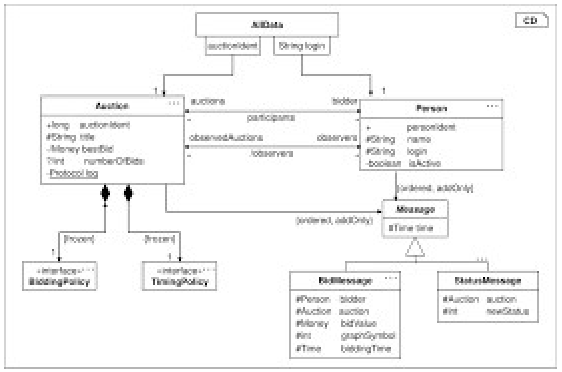


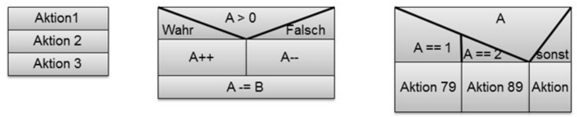
$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

```
funktion quicksort(links, rechts)
  falls links < rechts dann
    teiler := teile(links, rechts)
      quicksort(links, teiler-1)
      quicksort(teiler+1, rechts)
  ende
ende

funktion teile(links, rechts)
  i := links
  j := rechts - 1
  pivot := daten[rechts]
  wiederhole
    wiederhole solange daten[i] ...
      i := i + 1
    ende
    wiederhole solange daten[j] ...
      j := j - 1
    ende
    falls i < j dann
      tausche daten[i] mit daten[j]
    ende
  solange i < j
  falls daten[i] > pivot dann
    tausche daten[i] mit daten[r...
  ende
  antworte i
ende
```

```
funktion quicksort_iterativ(links,
rechts)
  wiederhole
    falls links < rechts dann
      pivot := daten[random(link...
      i := links-1
      j := rechts+1
      wiederhole
        wiederhole i := i+1 ...
        wiederhole j := j-1 ...
        falls i >= j dann Abbru...
          tausche daten[i],...
        ende
      push (rechts)
      rechts := max(links, i-1)
    sonst
      falls stapel leer dann ...
      links := rechts+1
      pop (rechts)
    ende
  ende
ende
```





Klassifikation von Algorithmen

- Einteilung der Algorithmen in Klassen
- Beschreibung der Algorithmen durch ihre Klassenmerkmale
- Klassifikationsmerkmale von Algorithmen:

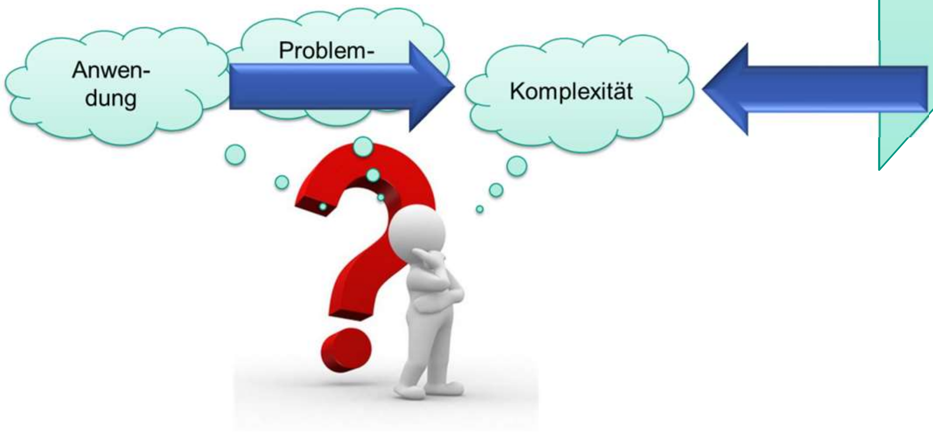
Klassen nach „Komplexität“

Platzkomplexität

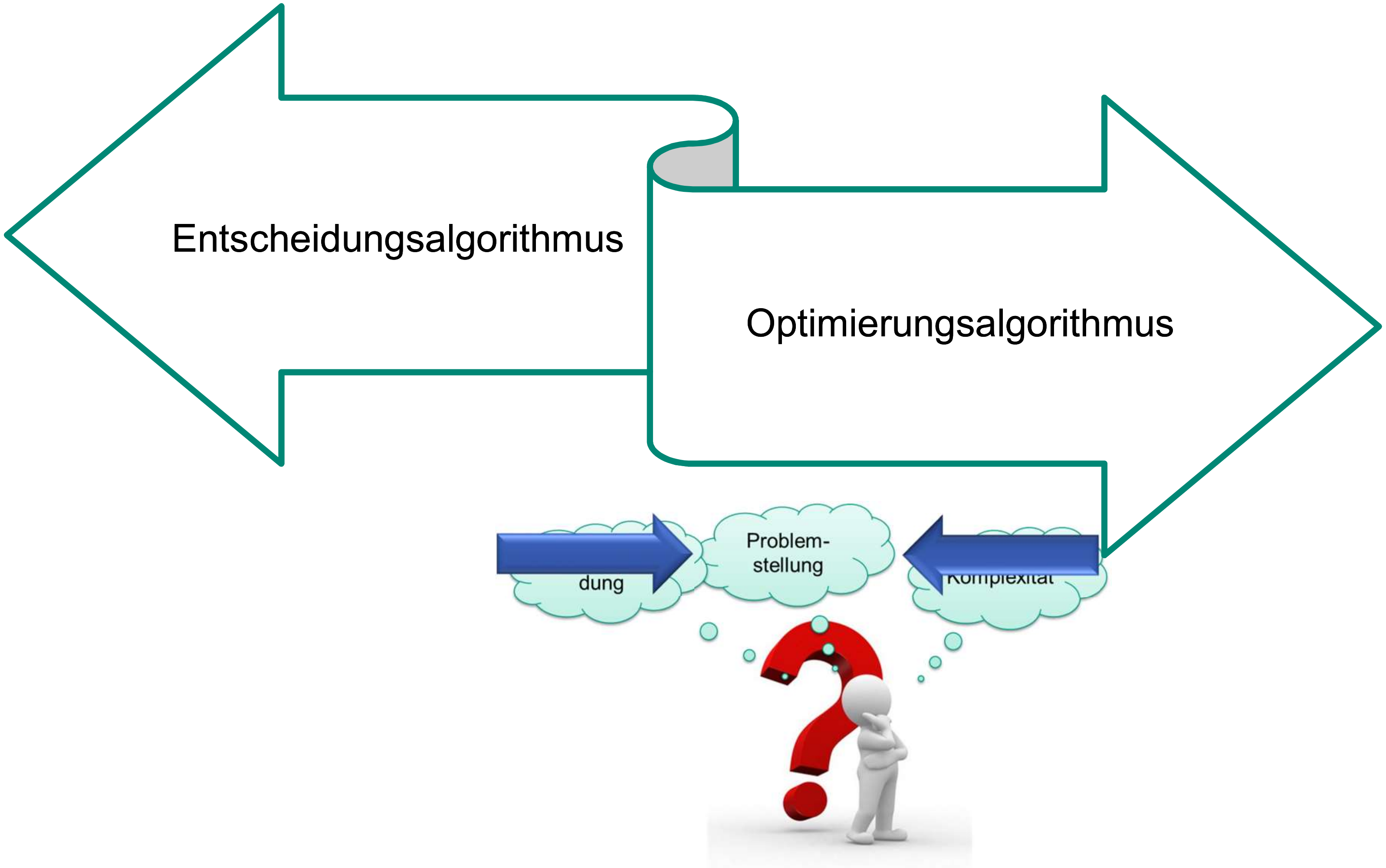
- Linear platzbeschränkt
- Logarithmisch platzbeschränkt
- Polynomial platzbeschränkt
- Exponentiell platzbeschränkt

Zeitkomplexität

- Linear zeitbeschränkt
- Logarithmisch zeitbeschränkt
- Polynomial zeitbeschränkt
- Exponentiell zeitbeschränkt

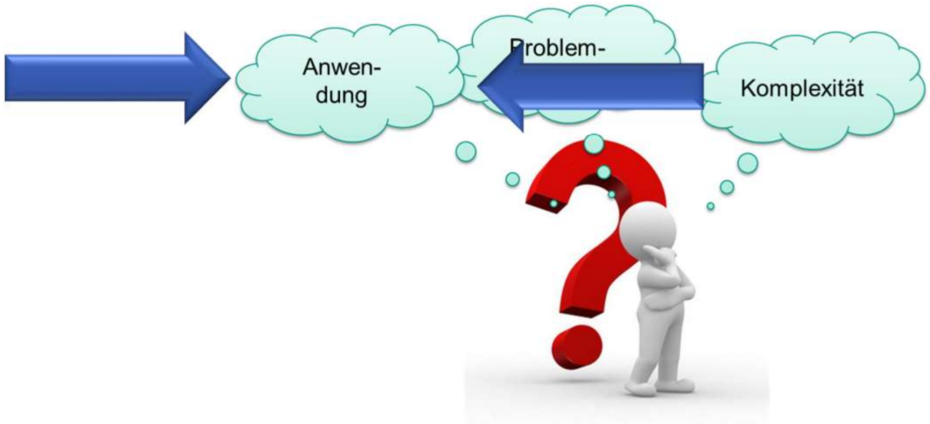


Klassen nach Problemstellung



Klassen nach Anwendung

- Suchalgorithmen
  - für Listen, Arrays: Lineare Suche, Binäre Suche, Interpolationssuche
  - für Graphen, Bäume: Breitensuche, Tiefensuche
- Sortieralgorithmen
  - Bubblesort, Heapsort, Insertionsort, Mergesort, Quicksort
- Graphentheorie
  - Kürzester Pfad, Breitensuche, Tiefensuche, Handlungsreisender
- Kryptographie: Verschlüsselungsalgorithmen
- Detektion: Bilderkennung
- ...



Weitere Merkmale von Algorithmen

Iterative Algorithmen

- Ein Algorithmus ist iterativ, wenn in seiner Realisierung keine Rekursionen vorkommen.

Rekursive Algorithmen

- Ein Algorithmus ist rekursiv, wenn dieser sich in seiner Realisierung nur selbst aufruft, jedoch keine Schleifen vorkommen.

Beispiel

Freie Universität Berlin

Rekursion vs. Iteration

Die Funktion **sum** berechnet für ein gegebenes **n>0** die Summe aller Zahlen von **1** bis **n**

Iterativ

```
def sum(n):
    sum = 0
    for i in range(n+1):
        sum += i
    return sum
```

1+2+3+...

Weitere Merkmale von Algorithmen

Iterative Algorithmen

- Ein Algorithmus ist iterativ, wenn in seiner Realisierung keine Rekursionen vorkommen.

Rekursive Algorithmen

- Ein Algorithmus ist rekursiv, wenn dieser sich in seiner Realisierung nur selbst aufruft, jedoch keine Schleifen vorkommen.

Beispiel

Freie Universität Berlin

Rekursion vs. Iteration

Die Funktion **sum** berechnet für ein gegebenes **n>0** die Summe aller Zahlen von **1** bis **n**

Iterativ

```
def sum(n):
    sum = 0
    for i in range(n+1):
        sum += i
    return sum
```

Rekursiv

```
def sum_rec(n):
    if n==0:
        return 0
    else:
        return n+sum_rec(n-1)
```

5+4+3+2+1

direkt

Formel von Gauß

```
def sum(n):
    return n*(n+1)//2
```

Weitere Merkmale von Algorithmen

Karlsruher Institut für Technologie

Iterative Algorithmen

- Ein Algorithmus ist iterativ, wenn in seiner Realisierung keine Rekursionen vorkommen.

Rekursive Algorithmen

- Ein Algorithmus ist rekursiv, wenn dieser sich in seiner Realisierung nur selbst aufruft, jedoch keine Schleifen vorkommen.

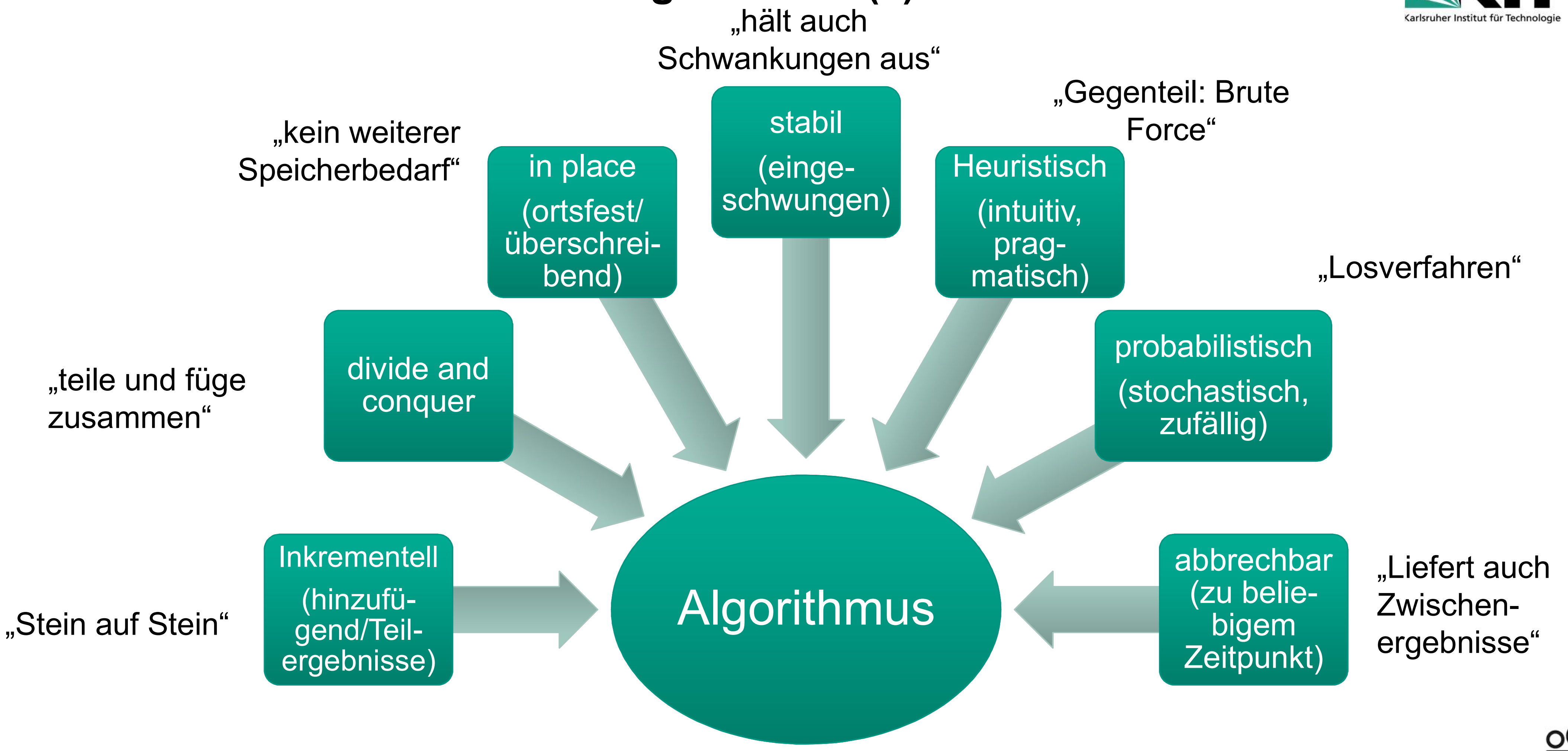
Halbiterative/ Halbrekursive Algorithmen

- Ein Algorithmus ist halbiterativ/ halbrekursiv, wenn er weder iterativ noch rekursiv ist, jedoch hauptsächlich solche iterative/ rekursive Merkmale besitzt.

Parallele Algorithmen

- Ein Algorithmus heißt parallel, wenn eine Laufzeitoptimierung möglich ist, indem er auf einem Mehrprozessorsystem implementiert wird.

Verschiedene Merkmale von Algorithmen (1)



<zur Erklärung der vorangegangenen Folie>

- **Inkrementell:** bedeutet im allgemeinen, dass der Algorithmus zunächst ein Teilergebnis bringt und dann während er läuft weitere Ergebnisse hinzufügt (inkrementell=hinzufügend).
  - Inkrementell kann auch bedeuten, dass ein Algorithmus neue Daten hinzufügend verarbeiten und das Ergebnis aktualisieren kann, ohne alles komplett neu zu berechnen.
- **divide and conquer**: Grundprinzip „teile und herrsche“
  - ein Problem wird solange in kleinere Teilprobleme zerlegt, bis man diese lösen (beherrschen) kann.
  - Dabei wird ausgenutzt, dass bei vielen Problemen der Lösungsaufwand sinkt, wenn man das Problem in kleinere Teilprobleme zerlegt → Anschließend wird aus den Teillösungen die Gesamtlösung (re)konstruiert.
- **in place:** Ein Algorithmus arbeitet in-place, wenn er außer dem für die Speicherung der zu bearbeitenden Daten benötigten Speicher nur eine konstante, also von der zu bearbeitenden Datenmenge unabhängige, Menge von Speicher benötigt.
  - Der Algorithmus überschreibt die Eingabedaten mit den Ausgabedaten. Man spricht von Ortsfestigkeit.
  - Das Gegenteil ist out-of-place.
- **stabil:** Ein Algorithmus sortiert stabil, wenn bei Gleichheit der Schlüssel 2er Elemente, die Reihenfolge der Elemente nicht verändert wird.
  - Ein (numerischer) Algorithmus heißt stabil wenn er für alle erlaubten und in der Größenordnung der Rechengenauigkeit gestörten Eingabedaten akzeptable Resultate produziert.

<zur Erklärung der vorangegangenen Folie>

- **heuristisch:** heuristische Algorithmen versuchen mit geringem Rechenaufwand und kurzer Laufzeit zulässige Lösungen für ein bestimmtes Problem zu erhalten.
  - Klassische Algorithmen versuchen, einerseits die optimale Rechenzeit und andererseits die optimale Lösung zu garantieren.
    - Heuristische Verfahren verwerfen einen oder beide dieser Ansprüche, um bei komplexen Aufgaben einen Kompromiss zwischen dem Rechenaufwand und der Güte der gefundenen Lösung einzugehen.
  - Dazu wird versucht, mithilfe von Schätzungen, „Faustregeln", intuitiv-intelligentem Raten oder unter zusätzlichen Hilfsannahmen eine gute Lösung zu erzeugen, ohne optimale Eigenschaften zu garantieren.
    - Heuristische Verfahren werden eingesetzt, wenn der erforderliche Rechenaufwand im Entscheidungsfindungsprozess zu umfangreich ist oder dieser den Rahmen des Möglichen sprengt.
    - Dabei wird die Anzahl der in Betracht zu ziehenden Möglichkeiten reduziert, indem man aussichtslos erscheinende Varianten von vornherein ausschließt.
  - Die Alternative zu heuristischen Verfahren ist die Brute-Force-Methode, bei der alle in Frage kommenden Möglichkeiten ausnahmslos durchgerechnet werden.
  - Die bekannteste und einfachste Heuristik ist die Lösung eines Problems mittels „Versuch und Irrtum" (Englisch: by trial and error).

<zur Erklärung der vorangegangenen Folie>

Karlsruher Institut für Technologie

- **probabilistisch:** Ein probabilistischer Algorithmus (auch stochastischer oder randomisierter Algorithmus) verwendet - im Gegensatz zu einem deterministischen Algorithmus - Zufallsbits um seinen Ablauf zu steuern.
  - Es wird nicht verlangt, dass ein randomisierter Algorithmus immer effizient eine richtige Lösung findet.
  - Randomisierte Algorithmen sind in vielen Fällen einfacher zu verstehen, einfacher zu implementieren und effizienter als deterministische Algorithmen für dasselbe Problem.
  - Wir unterscheiden drei verschiedene Arten von probabilistischen Algorithmen.
    - Algorithmus 1. Art (Macao Algorithmus):
      - mindestens bei einem Schritt der Prozedur werden einige Zahlen zufällig ausgewählt (nicht definit). Sonst deterministisch. Diese Algorithmen liefern immer eine korrekte Antwort. Benutzt werden sie, wenn irgendein bekannter Algorithmus zur Lösung eines bestimmten Problems im mittleren Fall viel schneller als im schlechtesten Fall läuft.
    - Algorithmus 2. Art (Monte-Carlo Algorithmus):
      - gleich wie Algorithmus 1. Art (nicht definit). Zusätzlich: Ausgabe ist korrekt mit einer Wahrscheinlichkeit von $1 - \varrho$, wobei ϱ sehr klein ist (nicht endlich). Diese Algorithmen liefern immer eine Antwort, wobei die Antwort nicht unbedingt richtig ist. $\varrho \to 0$ falls $t \to \infty$. Das Problem bei solchen Algorithmen liegt darin, zu entscheiden, ob die Antwort korrekt ist.
    - Algorithmus 3. Art (Las-Vegas Algorithmus):
      - gleich wie Macao-Algorithmus (nicht definit). Eine Folge von zufälligen Wahlen kann unendlich sein (mit einer Wahrscheinlichkeit $\varrho \to 0$) (nicht endlich). Diese Algorithmen liefern nie eine unkorrekte Antwort, jedoch besteht die Möglichkeit dass keine Antwort gefunden wird.
- **abbrechbar:** Algorithmus der so entworfen ist, dass er zu jedem beliebigen Zeitpunkt abbrechbar ist und das bis dahin erreichte Ergebnis ein gültiges (meist nicht optimales) Ergebnis ist, das sofort ausgeben werden kann.

Algorithmen-Analyse z.B. Laufzeit

- Die Erforschung und Analyse von Algorithmen ist eine Hauptaufgabe der Informatik
  - Dies wird meist theoretisch (ohne konkrete Umsetzung in eine Programmiersprache) durchgeführt.
- Algorithmen werden zur Analyse in eine stark formalisierte Form gebracht und mit den Mitteln der formalen Semantik untersucht.
- Schrittweises Vorgehen
  - Charakterisierung der Eingabedaten
  - Bestimmung der abstrakten Operationen
  - Iterative Verbesserung:

analysieren schätzen

verfeinern

- Problem für Laufzeitanalyse:
  - Abschätzung unter Verzicht auf Einzelheiten.
    - 90/10-Regel: 90% der Laufzeit bewirkt durch nur 10% des Programm-Codes
    - 1:50 Regel: 1% des Programm-Codes bewirkt 50% der Laufzeit

Teilgebiete der Algorithmenanalyse

- Komplexitätstheorie:
  - Verhalten von Algorithmen bezüglich Ressourcenbedarf wie Rechenzeit und Speicherbedarf
  - Ressourcenbedarf wird dabei in Abhängigkeit von der Länge der Eingabe ermittelt, z.B.:
    - Anzahl Elemente eines Feldes, einer Matrix
    - Länge einer Zeichenkette
    - Grad eines Polynoms
    - Anzahl der Knoten in einer dynamischen Datenstruktur (Liste, Baum)
  - → Die meisten Algorithmen besitzen einen Hauptparameter n, der die Anzahl der zu verarbeitenden Datenelemente angibt.
- Ziel: Bestimmung der Laufzeit unabhängig von
  - Hardware
  - Betriebssystem
  - Programmiersprache
  - Verwendete Bibliotheken
  - Compiler-Optimierungen

Relativer Zeitaufwand für „Integer“ Operationen
Feinanalyse anhand von Beispielen

Elementare Aktionen und Operationen mit *integer*-Datenobjekten und ihr relativer Zeitaufwand bezogen auf die Zuweisung

Elementare Aktion/Operation		Relativer Zeitaufwand
Zuweisung (= Basis)	=	1.0
Addition oder Subtraktion	+/-	1.4
Multiplikation	*	2.3
Division	/	8.0
Vergleich (inkl. Sprung bei *if* oder *while*)	<>	1.5
Indizierung einer Matrix	[ ]	4.2

Werte wurden ermittelt mit einem PASCAL-Programm,
Borland PASCAL-Compiler Version 7.0,
auf PC unter Betriebssystem Microsoft XP
mit Prozessor Intel Pentium 4 mit 2GHz Taktfrequenz

Relative Laufzeitberechnung

Beispiel: Array-Elemente initialisieren → Matrix[j,k] = 0

Algorithmus $A1$	Anzahl	Gewicht
`i := 1`	1	
`while i ≤ n1 do`	$n1 + 1$	
`  j := 1`	$n1$	
`  while j ≤ n2 do`	$n1 \cdot (n2 + 1)$	...
`    matrix[i, j] := 0.0`	$n1 \cdot n2$	
`    j := j + 1`	$n1 \cdot n2$	
`  end -- while`		
`  i := i + 1`	$n1$	
`end -- while`		
Summen		

=	1.0
+/-	1.4
*	2.3
/	8.0
<>	1.5
[ ]	4.2

Feinanalyse

Relative Laufzeitberechnung für Matrix[j,k] = 0

Algorithmus $A1$	Anzahl	Gewicht	Gesamt = Anzahl · Gewicht			
			$n1 \cdot n2$	$n1$	$n2$	konst.
`i := 1`	1	1.0				1.0
`while i ≤ n1 do`	$n1 + 1$	1.5		1.5		1.5
`j := 1`	$n1$	1.0		1.0		
`while j ≤ n2 do`	$n1 \cdot (n2 + 1)$	1.5	1.5	1.5		
`matrix[i, j] :=` 0.0	$n1 \cdot n2$	4.2	4.2			
`j := j + 1`	$n1 \cdot n2$	2.4	2.4			
`end -- while`						
`i := i + 1`	$n1$	2.4		2.4		
`end -- while`						
Summen			8.1	6.4	0.0	2.5

=	1.0
+/-	1.4
*	2.3
/	8.0
<>	1.5
[ ]	4.2

Feinanalyse

Relative Laufzeit Algorithmus 1: $t_{A1}(n1, n2) = 8{,}1 \cdot n1 \cdot n2 + 6{,}4 \cdot n1 + 0 \cdot n2 + 2{,}5$

Relative Laufzeitberechnung für Matrix[j,k] = 0

Alternative Realisierung

Algorithmus $A2$	Anzahl	Gewicht	Gesamt = Anzahl · Gewicht $n1 \cdot n2$	$n1$	$n2$	konst.
`l := 1`	1	1.0				1.0
`c := 1`	1	1.0				1.0
`i := 1`	1	1.0				1.0
`noe := n1 * n2`	1	3.3				3.3
`while i ≤ noe do`	$n1 \cdot n2 + 1$	1.5	1.5			1.5
`matrix[l, c] :=` **0.0**	$n1 \cdot n2$	4,2	4.2			
`if c = n2 then`	$n1 \cdot n2$	1.5	1.5			
`l := l + 1`	$n1$	2.4		2.4		
`c := 1`	$n1$	1,0		1.0		
`else`						
`c := c + 1`	$n1 \cdot n2 - n1$	2.4	2.4	−2.4		
`end -- if`						
`i := i + 1`	$n1 \cdot n2$	2.4	2.4			
`end -- while`						
Summen			12.0	1.0	0.0	7.8

=	1.0
+/-	1.4
*	2.3
/	8.0
<>	1.5
[ ]	4.2

Feinanalyse

Relative Laufzeit Algorithmus 2: $t_{A2}(n1, n2) = 12{,}0 \cdot n1 \cdot n2 + 1{,}0 \cdot n1 + 0 \cdot n2 + 7{,}8$

Vergleich Algorithmus 1 und 2

- Laufvariable = Indexvariable
- Geschachtelte While-Schleife

- Laufvariable ≠ Indexvariable
- 1 While-Schleife und
  1 if/else-Abfrage

Algorithmus A1

```
i := 1
while i ≤ n1 do
  j := 1
  while j ≤ n2 do
    matrix[i, j] := 0.0
    j := j + 1
  end -- while
  i := i + 1
end -- while
Summen
```

```
l := 1
c := 1
i := 1
noe := n1 * n2
while i ≤ noe do
  matrix[l, c] := 0.0
  if c = n2 then
    l := l + 1
    c := 1
  else
    c := c + 1
  end -- if
  i := i + 1
end -- while
Summen
```

Vergleich der relativen Laufzeiten

- Algorithmus 1 $$t_{A1} = 8{,}1 \cdot n1 \cdot n2 + 6{,}4 \cdot n1 + 2{,}5$$
- Algorithmus 2 $$t_{A2} = 12{,}0 \cdot n1 \cdot n2 + 1{,}0 \cdot n1 + 7{,}8$$
- Zum einfacheren Vergleich des Laufzeitverhaltens
  - Einführung neue Problemgröße:
  - grobe Maßzahl für die Anzahl der Durchläufe

$$\Rightarrow n = \sqrt{n1 \cdot n2}$$

- … und für großes n: $$t'_{A1} = 8{,}1 \cdot n^2 \qquad t'_{A2} = 12{,}0 \cdot n^2$$
- Analyseergebnis:
  - Algorithmus 1 und 2 zeigen ein quadratisches Wachstum mit n.
  - Algorithmus 2 braucht zum Lösen der Aufgabe für große n (> 2) etwa 50% mehr Rechenzeit
- Überprüfung mit Zeitmessung auf genanntem Rechner
  - für n1 = n2 = n = 100 und 10.000 Wiederholungen
  - Bei 10.000 Wdh. → Laufzeit 1,37 Sekunden bzw. 2,09 Sekunden

Fragen?

Laufzeitkomplexität → Grobanalyse

- Feinanalyse (s. zuvor) wird i.A. nur bei sicherheitsrelevanten Systemen mit harten Realzeitanforderungen durchgeführt
- In den meisten praktischen Anwendungen nur Grobanalyse
  - Identifiziere und analysiere die Teile des Algorithmus, die das Laufzeitverhalten signifikant beeinflussen
  - Suche innerste (am häufigsten zu durchlaufende) Schleife
  - Analysiere prinzipielles Laufzeitverhalten in Abhängigkeit der Problemgröße n
  - Analysiere Abhängigkeit von Inhalt und Ausprägung der verarbeitenden Datenobjekte:
    - Günstigster Fall (best case) → minimale Zahl an Arbeitsschritten
    - Ungünstigster Fall (worst case) → maximale Zahl an Arbeitsschritten
    - Durchschnittlicher Fall (average case)
    - ► typische, über viele Anwendungen hinweg betrachtete und gemittelte Zahl an Arbeitsschritten
    - ► häufig nur auf Basis von Annahmen und mit Hilfe Wahrscheinlichkeitsrechnung

Effizienz versus Effektivität

- Effektivität: Wirksamkeit des Algorithmus zur möglichst guten Lösung der Aufgabenstellung unabhängig vom Aufwand
- Effizienz: Wirtschaftlichkeit des Algorithmus.

  Verhältnis von Aufwand zum Grad der Zielerreichung

  → Effizienz setzt Effektivität voraus

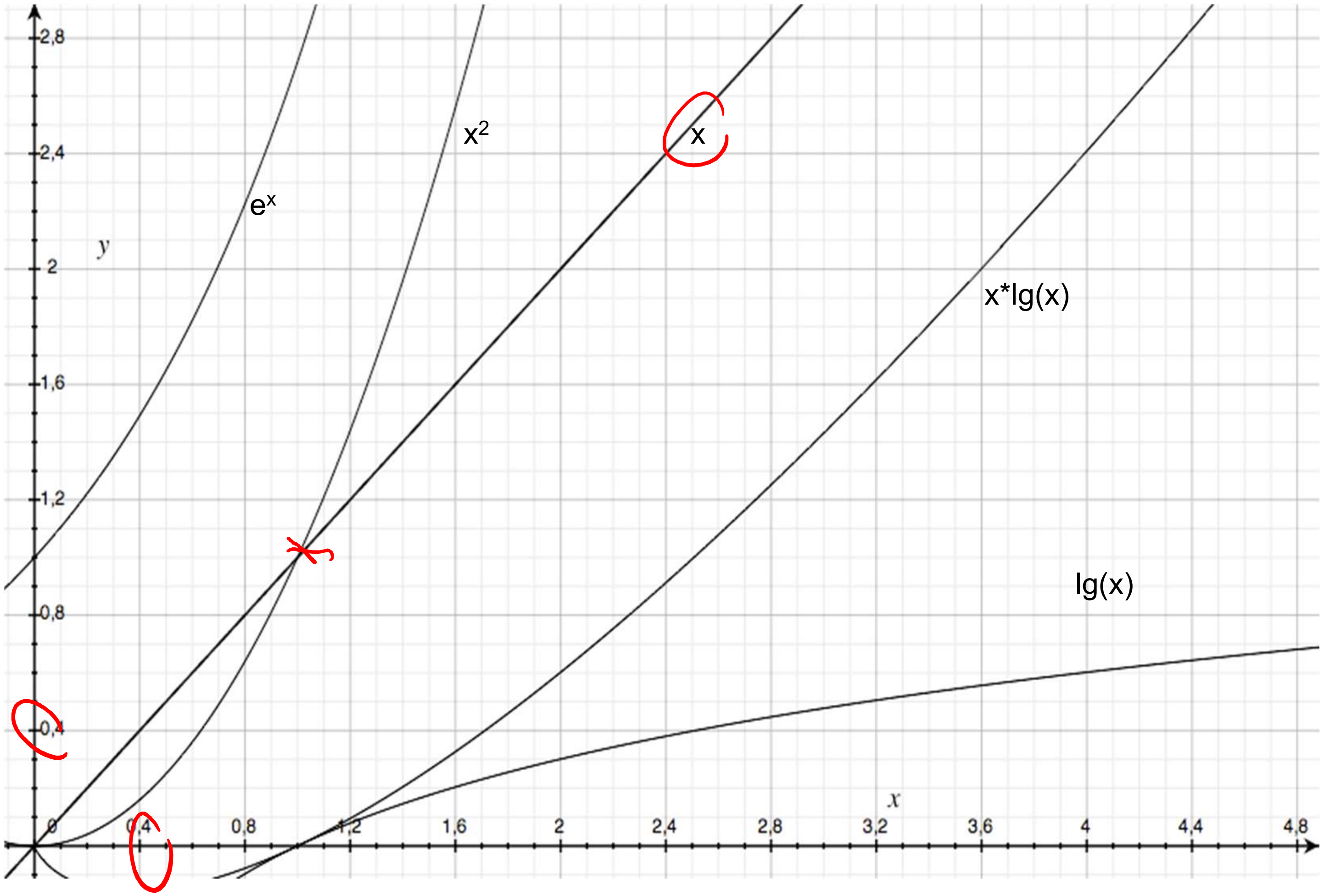
O-Notation

- Ursprung
  - Die *O*-Notation (gesprochen: „groß oh“, als Symbol für „*Ordnung von“*) wird grundsätzlich auf die Veröffentlichung *Analytische Zahlentheorie* aus dem Jahr 1892 von Peter Bachmann (1837 - 1920) zurückgeführt.
  - Einige Zeit später hat der Zahlentheoretiker Edmund Landau (1877 -1938) davon Gebrauch gemacht und neben ‚*O*’ auch ‚*o*’ betrachtet.
  - Aufgrund dessen werden Notationen auch als Gebrauch Landau’scher Symbole bezeichnet.
  - Remark: *Auf Landau geht auch das Symbol für die Menge der ganzen Zahlen zurück.*
- Zweck
  - Die *O*-Notation wird genutzt, um Abschätzungen über Aufwände von Algorithmen vorzunehmen.
  - obere Schranken erkennen, da Probleme vorwiegend durch die hohen Aufwände für Speicher und Laufzeit entstehen.
  - Ermittlung unterer Schranken und arithmetischer Mittel.
    - Untere Schranken können von Bedeutung sein, um zu zeigen, dass es keinen Algorithmus gibt, der schneller läuft als $g(n)$.

Laufzeitkomplexitätsklassen

Asymptotische Laufzeit-komplexität	Bezeichnung	Erläuterung am Beispiel der Verdoppelung der Problemgröße und typische Algorithmen mit dieser Ordnung
$O(1)$	konstant	Laufzeit ist unabhängig von der Problemgröße, optimaler Fall, der praktisch nicht auftritt
$O(log\ n)$	logarithmisch	Verdoppelung der Problemgröße bewirkt Anstieg der Laufzeit um *log* 2, also um eine Konstante (um 1 für *logarithmus dualis*), sehr günstig und daher erstrebenswert, z.B. *binäre Suche* (siehe Kapitel 6)
$O(n)$	linear	Verdoppelung der Problemgröße bewirkt Verdoppelung der Laufzeit, immer noch zufrieden stellend, z.B. *sequenzielle Suche* (siehe Kapitel 6)
$O(n\ log\ n)$	–	Fast so gut wie linear, v hältnis zu n klein ist, z.B. gute Sortierverfahren wie *Quicksort* (siehe Kapitel 7)
$O(n^2)$	quadratisch	Verdoppelung der Problemgröße bewirkt Vervierfachung der Laufzeit, ungünstig, z.B. schlechte Sortierverfahren wie *Bubblesort* (siehe K
$O(n\ log\ n)$	–	Fast so gut wie linear, weil *log* n im Verhältnis zu n klein ist, z.B. gute Sortierverfahren wie *Quicksort*
$O(n^2)$	quadratisch	Verdoppelung der Problemgröße bewirkt Vervierfachung der Laufzeit, ungünstig, z.B. schlechte Sortierverfahren wie *Bubblesort*
$O(n^3)$	kubisch	Verdoppelung der Problemgröße bewirkt Verachtfachung der Laufzeit, sehr unbefriedigend, z.B. einfache *Matrizenmultiplikation*
$O(k^n)$	exponentiell	Verdoppelung der Problemgröße bedeutet Quadrierung (weil $k^{2n} = (k^n)^2$) der Laufzeit, katastrophal, z.B. *Back-tracking*- oder *Exhaustionsalgorithmen*

Beispiel: Komplexitätsverhalten (kleine n)



Zwischenübung: Typische Wachstumsgesetze

- Annahme: Verwendet wird ein Rechner mit 10^9 Instruktionen pro Sekunde (1.000 MIPS) und ein Algorithmus mit 1.000 Instruktionen.
- Wieviele Algorithmen mit Wachstum g(n) (s. Zeilen) können in der Zeit t (s. Spalten) umgesetzt werden?

g(n)	1 Sekunde	1 Minute	1 Stunde	1 Tag	1 Monat	1 Jahr	1 Jahrhundert
$\log_2 n$							
$\sqrt{n}$		$= 10^9 \div 10^3 = 10^6 = \sqrt{n} \rightarrow n = 10^{12}$					
n		$= 10^9 \div 10^3 = 10^6 = n$					
$n \log_2 n$							
n^2		$= 10^9 \div 10^3 = 10^6 = n^2 \rightarrow n = 10^3$					
n^3							
2^n							
$n!$							

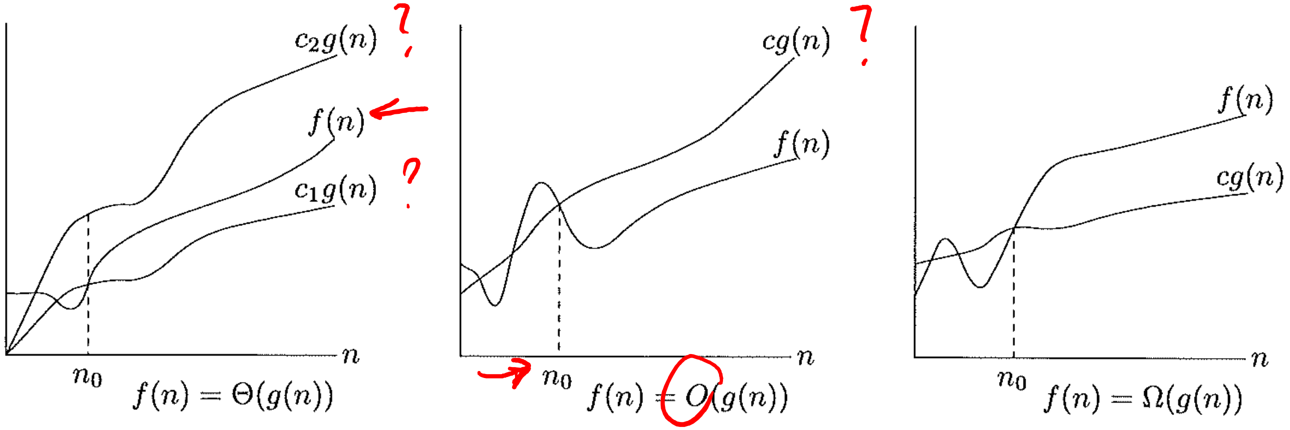
Zwischenübung: Typische Wachstumsgesetze Lsg.

- Annahme: Verwendet wird ein Rechner mit 10^9 Instruktionen pro Sekunde (1.000 MIPS) und ein Algorithmus mit 1.000 Instruktionen.
- Wieviele Algorithmen mit Wachstum g(n) (s. Zeilen) können in der Zeit t (s. Spalten) umgesetzt werden?

g(n)	1 Sekunde	1 Minute	1 Stunde	1 Tag	1 Monat	1 Jahr	1 Jahrhundert
$\log_2 n$	2^{10^6}	$\approx\infty$	$\approx\infty$	$\approx\infty$	$\approx\infty$	$\approx\infty$	$\approx\infty$
$\sqrt{n}$	1E12	$= 10^9 \div 10^3 = 10^6 = \sqrt{n} \rightarrow n = 10^{12}$					9,9E30
n	1E6	$= 10^9 \div 10^3 = 10^6 = n$			2,7E12	31,5E12	3,15E15
$n\log_2 n$	62746	2,8E6	1,3E8	2,8E9	7,5E10	8,0E11	6,9E13
n^2	1000	$= 10^9 \div 10^3 = 10^6 = n^2 \rightarrow n = 10^3$				E6	5,6E7
n^3	100	391	1532	4420	13924	31581	146589
2^n	19	$= 2^{20} = 1.048.576$		36	41	44	51
$n!$	9	11	12	13	15	16	17

Notationen zur Definition der Ordnung bzw. von Schranken für das Wachstum von Funktionen

Notation	Definition	Interpretation
O-Notation	$f(n) = O(g(n))$ bedeutet $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$	f ist *höchstens* von der Ordnung g g definiert obere Schranke für f
Ω-Notation	$f(n) = \Omega(g(n))$ bedeutet $\lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c$	f ist *mindestens* von der Ordnung g g definiert untere Schranke für f
Θ-Notation	$f(n) = \Theta(g(n))$ bedeutet $c_1 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c_2$	f ist *genau* von der Ordnung g g definiert eine Bandbreite für f

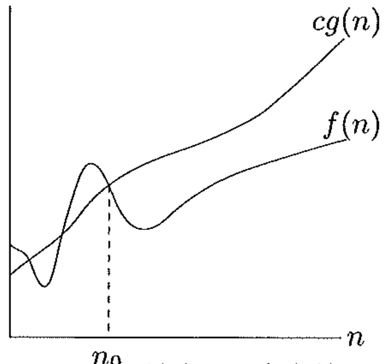


O-Notation: asymptotische Laufzeitkomplexität

- Obere asymptotische Schranke
- Bei einer gegebenen Funktion g(n) bezeichnen wir mit O(g(n)) (ausgesprochen „groß O von g von n“) die Menge der Funktionen

  O(g(n)) = {f(n), so dass für positive Konstanten c und n_0 gilt

  $0 < f(n) < cg(n)$ für alle $n > n_0$ }



$f(n) = O(g(n))$

gesucht :
- möglichst “kleine” und “einfache” Funktionen g(n)
- mit möglichst geringen konstanten Faktoren c
- als asymptotische obere Schranke für f(n)

Beispiel asymptotische Laufzeitkomplexität

Karlsruher Institut für Technologie

- Ermittlung der asymptotischen Laufzeitkomplexität des Algorithmus A1

$$t_{A1}(n1, n2) = 8{,}1 \cdot n1 \cdot n2 + 6{,}4 \cdot n1 + 2{,}5$$

- o.B.d.A. mit Problemgröße $n1 = n2 = n$

$$t_{A1}(n) = 8{,}1 \cdot n^2 + 6{,}4 \cdot n + 2{,}5 = n^2 \cdot (8{,}1 + 6{,}4 / n + 2{,}5 / n^2)$$

O(g(n)) = {f(n), so dass für positive Konstanten c und n_0 gilt

0 < f(n) < cg(n) für alle n > n_0 }

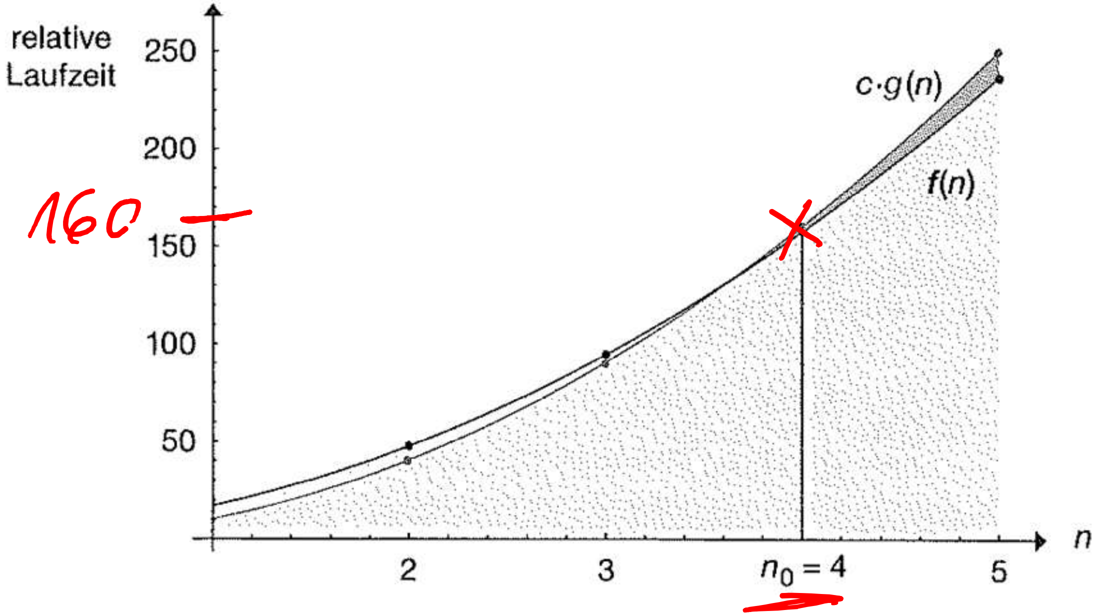
- Daraus ist ersichtlich, dass der Wert der Konstanten c gemäß Definition der O-Notation auf jeden Fall größer als 8,1 sein muss.

Beispiel asymptotische Laufzeitkomplexität

$$f(n) = n^2 \cdot (8{,}1 + 6{,}4/n + 2{,}5/n^2)$$
$$g(n) = n^2$$

- Für diese Funktion f(n) gilt also schon für

$$n_0 = 4,\ \text{dass}\ f(n) \le c \cdot g(n)\ \text{mit}\ c = 10$$
$$und\ \ g(n) = n^2\ \text{ist.}$$



Ende VL2