Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Vorlesung: Informationstechnik (IT)

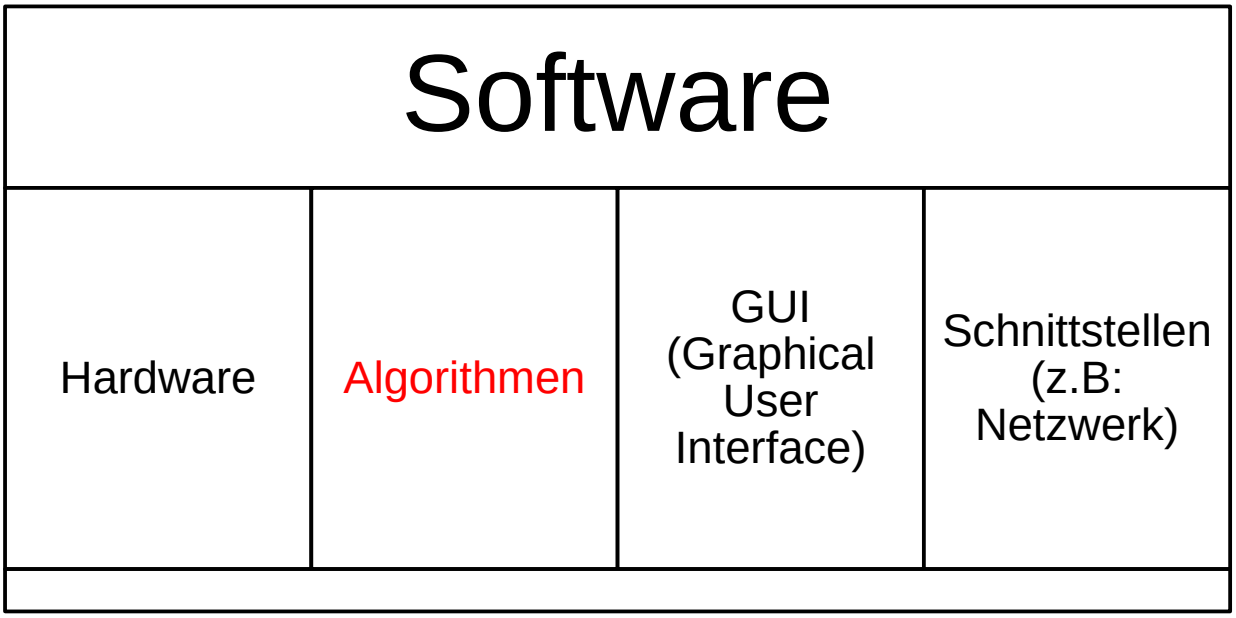
Prof. Dr.-Ing. K.D. Müller-Glaser

Institut für Technik der Informationsverarbeitung (ITIV)

Algorithmen

Algorithmen in Software

- Algorithmen sind ein Basisbestandteil von Software:



Algorithmen stellen eine Basis-Technologie dar, mit gleicher Wichtigkeit wie andere Computertechnologien (Hardware, Vernetzung, GUI etc.).

Definition: Algorithmus

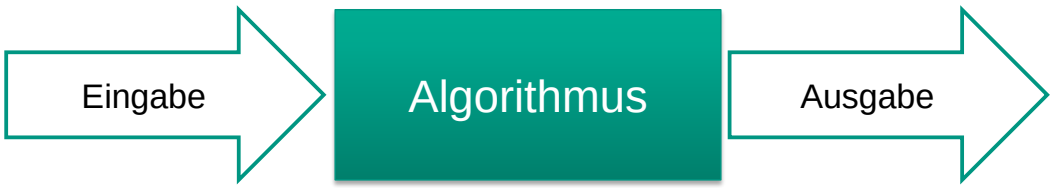
- Genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten

- Beispiele:
  - Kochrezept
  - Reparatur- und Gebrauchsanleitungen
  - Hilfen zum Ausfüllen von Formularen
  - Waschmaschinenprogramme

Das Wort Algorithmus: Abwandlung oder Verballhornung des Namens von Muhammed Al Chwarizmi (ca. 783-850), dessen arabisches Lehrbuch (um 825) „Über das Rechnen mit indischen Ziffern“ in der mittelalterlichen lateinischen Übersetzung mit den Worten: „Dixit Algorismi“ (Algorismi hat gesagt) begann.

Definition: Algorithmus

- Algorithmus im engeren Sinne der Informatik
  - Gegenstand einiger Spezialgebiete der Theoretischen Informatik (Algorithmentheorie, Komplexitätstheorie, Berechenbarkeitstheorie)
  - In Form von Computerprogrammen und elektronischen Schaltkreisen steuern sie Computer und andere Maschinen



- Wohldefinierte Rechenvorschrift, die eine Größe oder eine Menge von Größen als Eingabe verwendet und eine Größe oder eine Menge von Größen als Ausgabe erzeugt.
- 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:
  - Finitheit: das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein
  - Ausführbarkeit: jeder Schritt des Verfahrens muss tatsächlich ausführbar sein
  - Dynamische Finitheit: das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen (Platzkomplexität)
  - Terminierung: 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 denselben Voraussetzungen das gleiche Ergebnis liefern
  - Determinismus: die nächste anzuwendende Regel im Verfahren ist zu jedem Zeitpunkt eindeutig definiert

Karlsruher Institut für Technologie

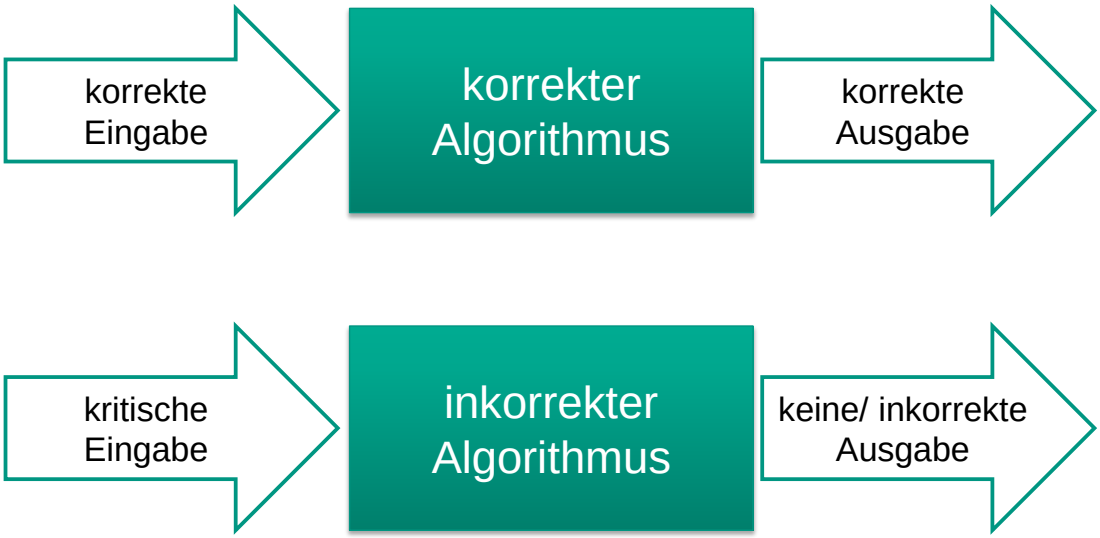
Algorithmus

Datenstrukturen

- Nach moderner Auffassung sind Datenstrukturen Bestandteil von Algorithmen.
- Bei vielen Algorithmen hängt der Ressourcenbedarf, also sowohl die benötigte Laufzeit als auch der Speicherplatzbedarf, von der Verwendung geeigneter Datenstrukturen ab.

Eine Datenstruktur ist eine Methode, Daten abzuspeichern und zu organisieren sowie den Zugriff auf die Daten und die Modifikation der Daten zu erleichtern. Keine Datenstruktur arbeitet für alle Zwecke gleich gut, deshalb ist es wichtig, Vorteile und Einschränkungen jeder Datenstruktur zu kennen.

Korrekte Algorithmen



Ein Algorithmus wird als korrekt bezeichnet, wenn er für jede Eingabeinstanz mit der korrekten Ausgabe stoppt (man sagt: ein korrekter Algorithmus löst ein gegebenes Rechenproblem).
Ein inkorrekter Algorithmus kann bei einigen Eingabeinstanzen überhaupt nicht stoppen, oder er stoppt mit einem anderen als dem gewünschten Ergebnis.

Beschreibung von Algorithmen

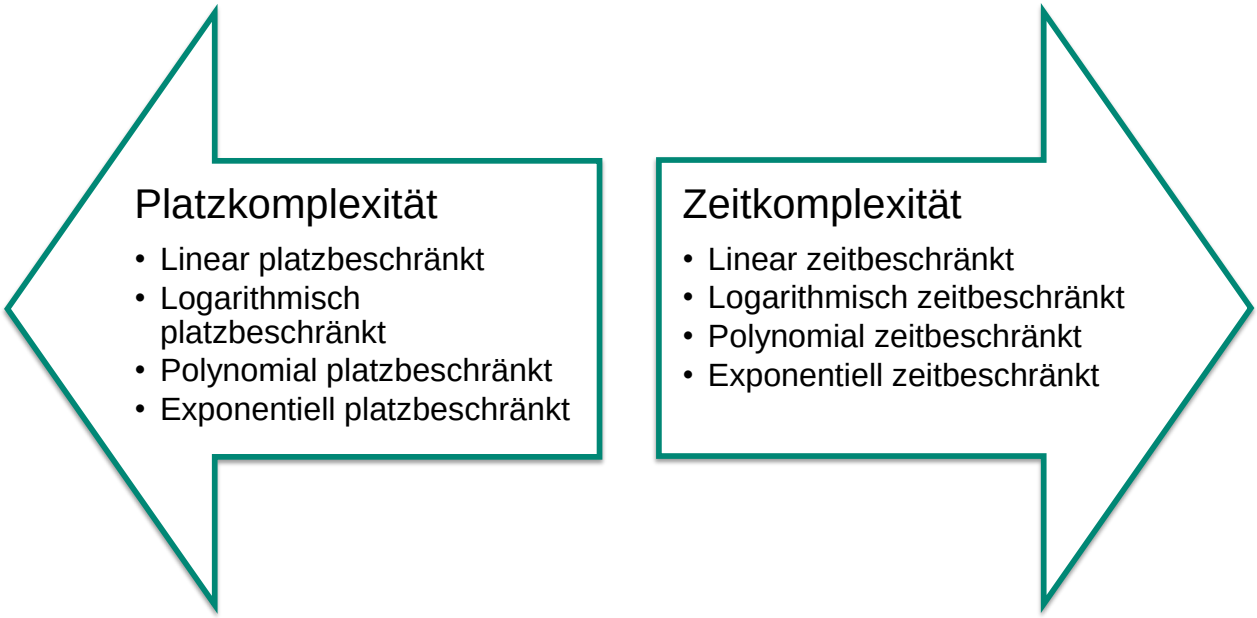
- Lösung eines Problems: Finden einer algorithmischen Beschreibung des Lösungswegs
- Beschreibung mittels:
  - der natürlichen Sprache
  - als Computerprogramm
  - als Hardwareentwurf
  - mit Pseudo Code
  - grafisch, z.B: Nassi Shneiderman Diagrammen

Klassifikation von Algorithmen

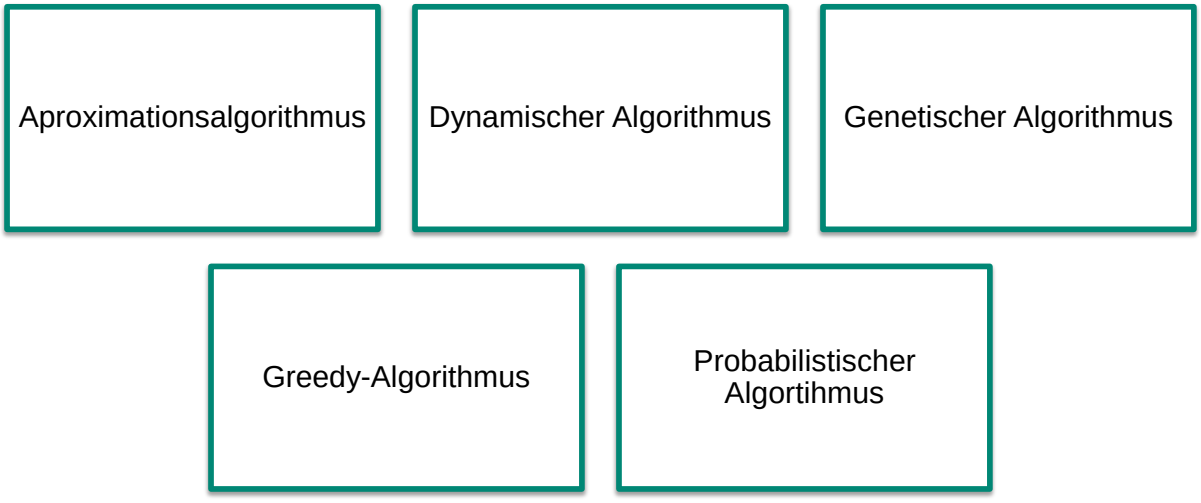
- Einteilung der Algorithmen in Klassen
- Beschreibung der Algorithmen durch ihre Klassenmerkmale

- Klassifikationsmerkmale von Algorithmen:
  - Komplexität
  - Verfahren
  - Problemstellung
  - Anwendung

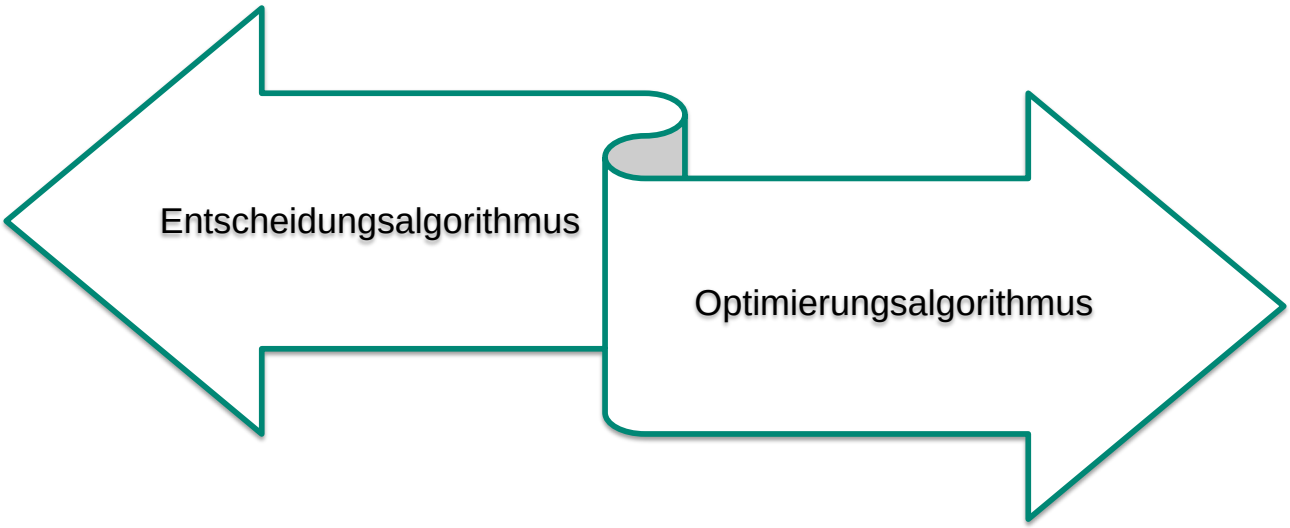
Klassen nach Komplexität



Klassen nach Verfahren



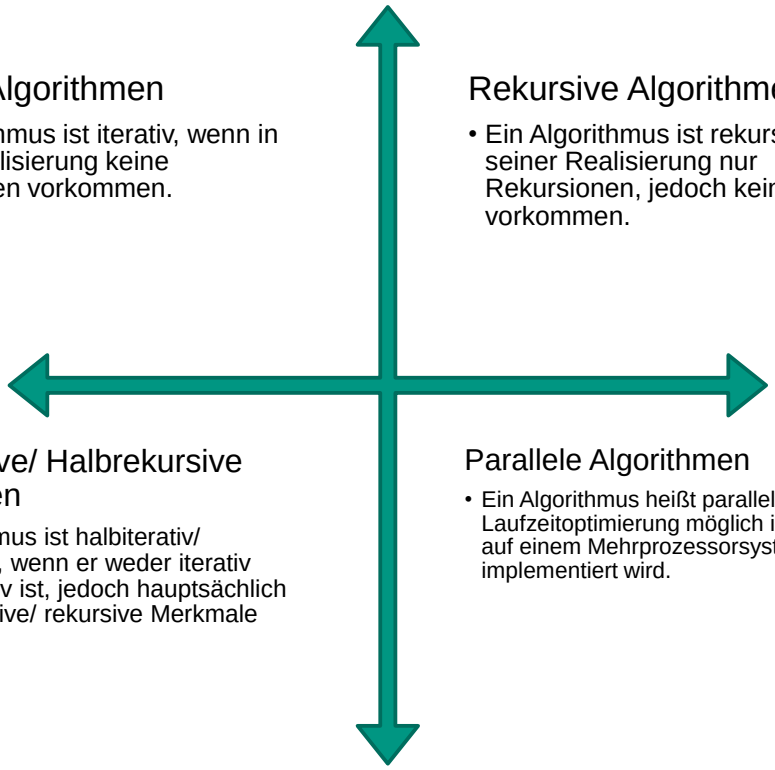
Klassen nach Problemstellung



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
- Klassifikation: 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 in seiner Realisierung nur Rekursionen, 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.

Iterative Algorithmen (vollständig):

Ein Algorithmus ist iterativ, wenn in seiner Realisierung (Programmcode) keine Rekursionen vorkommen.

Rekursive Algorithmen (vollständig):

Ein Algorithmus ist rekursiv, wenn in seiner Realisierung (Programmcode) nur Rekursionen (direkt oder indirekt), jedoch keine Schleifen (for, while, repeat, ...) vorkommen. Zu jedem iterativen Algorithmus kann ein äquivalenter rekursiver Algorithmus angegeben werden.

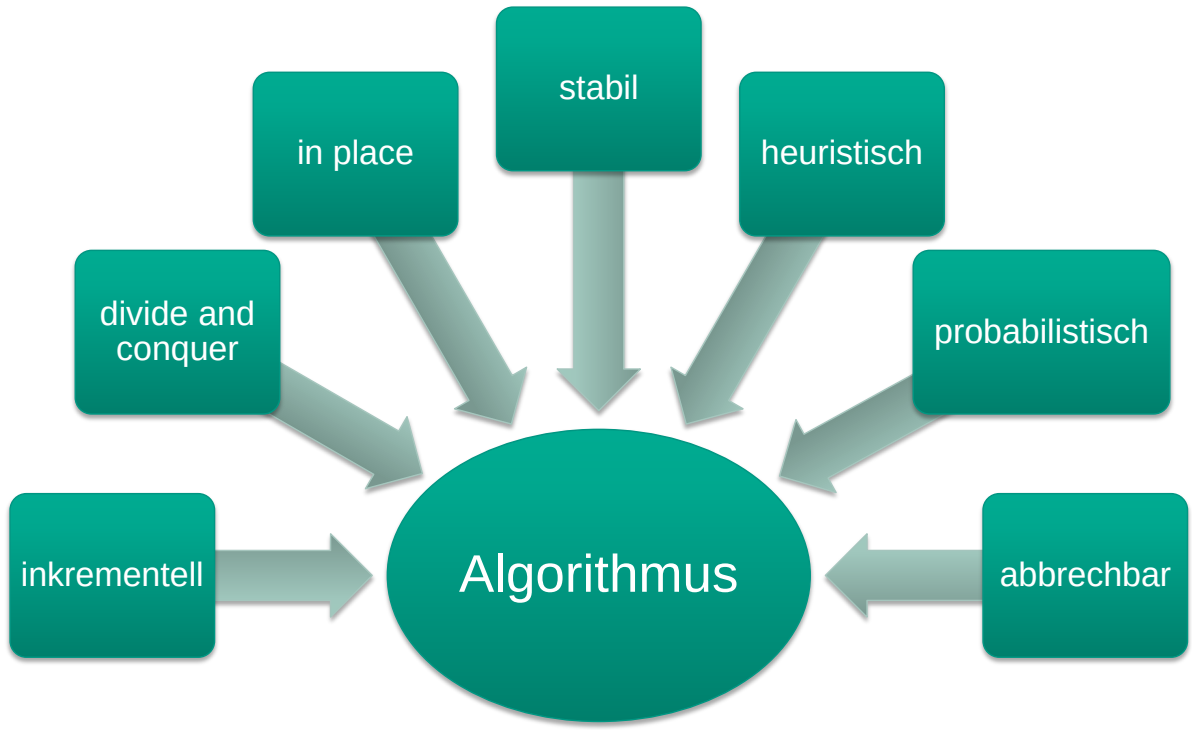
Halbiterative/Halbrekursive Algorithmen:

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

Parallele Algorithmen:

Ein Algorithmus heißt parallel (oder parallelisierbar), wenn eine Laufzeitoptimierung möglich ist, indem er auf einem Multiprozessorsystem implementiert wird. D.h. Teilaufgaben können von mehreren Prozessoren parallel abgearbeitet werden.

Weitere Merkmale von Algorithmen



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.
Ansschliessend wird aus den Teillösungen die Gesamtlösung (re)konstruiert.

in place: Ein Algorithmus arbeitet in-place bzw. in situ, 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. Hierbei ist auch von Ortsfestigkeit die Rede. Das Gegenteil ist out-of-place.

stabil: Ein Algorithmus sortiert stabil, wenn bei Gleichheit der Schlüssel zweier 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.

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).

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.

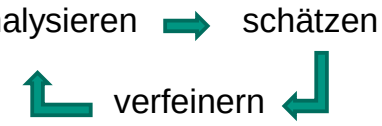
abbrechbar: Algorithmus der so entworfen ist, dass er zu jedem beliebigen Zeitpunkt abbrechbar ist und das bis dahin erreichte Ergebnis ein gültiges (meist nicht optimales) Ergebnis ist, das sofort ausgeben werden kann.

Algorithmenanalyse

- Die Erforschung und Analyse von Algorithmen ist eine Hauptaufgabe der Informatik, diese 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
  1. Charakterisierung der Eingabedaten
  2. Bestimmung der abstrakten Operationen
  3. Iterative Verbesserung:



  4. Konzentration auf die laufzeitbestimmenden „innersten Schleifen“ 90/10-Regel und 1:50 Regel

Charakterisieren der Eingabedaten: ideal: für beliebige Wahrscheinlichkeitsverteilungen Herleitung der Laufzeiten des Algorithmus,

da praktisch oft nicht möglich, Schranken für Leistungskenngrößen suchen (siehe auch O-Notation): Laufzeit stets kleiner als gewisse „obere Schranke“ unabhängig von Eingabedaten
bzw. durchschnittliche Laufzeit für „zufällige“ Eingaben.

Abstrakte Operationen: dienen zur Trennung der Analyse von der Implementierung

z.B Anzahl der Vergleiche in einem Sortieralgorithmus versus Anzahl der Mikrosekunden, die ein bestimmter Mikrocomputer zur Ausführung des Maschinencodes if a[i] > v benötigt.

Beide Komponenten werden gebraucht, um die tatsächliche Laufzeit eines Programmes auf einem speziellen Computer zu ermitteln.

Die erste ist bestimmt durch die Eigenschaften des Algorithmus,

die zweite ist bestimmt durch die Eigenschaften des Computers.

Primär bei Algorithmenentwicklung und -vergleich soll von Hardware abstrahiert werden.

Problem für Laufzeitanalyse: finde nicht nur eine obere Schranke, sondern die „beste“ obere Schranke.

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,
somit wegen 90/10 Regel weitere 9% des Codes bewirken weitere 40% der Laufzeit.

Teilgebiete der Algorithmenanalyse

- Komplexitätstheorie:
  - Verhalten von Algorithmen bezüglich Ressourcenbedarf wie Rechenzeit und Speicherbedarf
  - Analyseergebnisse werden als asymptotische Laufzeiten angegeben
  - Ressourcenbedarf wird dabei in Abhängigkeit von der Länge der Eingabe ermittelt
    - Anzahl Elemente eines Feldes, einer Matrix
    - Länge einer Zeichenkette
    - Grad eines Polynoms
    - Anzahl der Knoten in einer dynamischen Datenstruktur (Liste, Baum)

- Berechenbarkeitstheorie:
  - Das Verhalten bezüglich der Terminierung, ob also der Algorithmus überhaupt jemals beendet werden kann

Effizienz versus Effektivität

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

- Aufwandsbetrachtung:
  - Strukturkomplexität (Algorithmus verstehen und testen)
  - Speicherkomplexität (Speicherplatzbedarf)
  - Laufzeitkomplexität (Rechenzeitbedarf)

- Effizienz: Wirtschaftlichkeit des Algorithmus.
  Verhältnis von Mitteleinsatz zum Grad der Zielerreichung

  Effizienz setzt Effektivität voraus

Effizienz eines Algorithmus

- Verschiedene Algorithmen zur Lösung des gleichen Problems unterscheiden sich stark in Bezug auf Speicherplatzbedarf und Rechenzeit
- Unterschiede in der Komplexität zweier Algorithmen können erheblicher sein, als durch Hardware und Software bedingte Unterschiede
- Ziel: Bestimmung der Laufzeit unabhängig von
  - Hardware
  - Betriebssystem
  - Programmiersprache
  - Verwendete Bibliotheken
  - Compileroptimierungen

Relativer Zeitaufwand für Integer Operationen Feinanalyse

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 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

Feinanalyse

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

Relative Laufzeitberechnung für Matrix[j,k] = 0

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

Feinanalyse

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

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 $n = \sqrt{n1 \cdot n2}$
und für großes n

$$t'_{A1} = 8{,}1 \cdot n^2$$
$$t'_{A2} = 12{,}0 \cdot n^2$$

Analyseergebnis:
Algorithmus 1 und 2 zeigen ein quadratisches Wachstum mit n.
Algorithmus 2 braucht zum Lösen der Aufgabe für große n etwa 50% mehr Rechenzeit

(Überprüfung mit Zeitmessung auf oben genanntem Rechner für n1=n2=n=100
und zehntausend Wiederholungen ergab Laufzeit 1,37 Sekunden bzw. 2,09 Sekunden)

Laufzeitkomplexität Grobanalyse

- Feinanalyse 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 mithilfe Wahrscheinlichkeitsrechnung)

Beschreibung der Effizienz

- Problem:
  - Vergleichbare Beschreibung der Effizienz verschiedener Algorithmus auf unterschiedlichen Rechnern, bei variabler Eingabe
- Folgende Annahmen:
  - Algorithmus besitzt Instruktionen
  - Bestimmte Instruktion kostet Faktor c an Ressourcen (Zeit/Speicher)
  - Eingabe umfasst n Elemente
- Resultat:
  - Wachstumsgesetz für den besten Fall (best case)
  - Wachstumsgesetz für den durchschnittlichen Fall (average case)
  - Wachstumsgesetz für den schlechtesten Fall (worst case)

(Beispiel hierzu siehe Kapitel Sortieralgorithmen)

Kostenfunktionen hängen vom Maschinenmodell ab. Im Allgemeinen ist es nicht möglich, für jede Art von Eingabe eine Wachstumsfunktion zu formulieren.

Effizienz von Algorithmen

Effizienz von Algorithmen hinsichtlich Rechenzeit kann erheblicher sein als durch Hardware und Software bedingte Unterschiede

Beispiel Rechenzeit für Sortieralgorithmus (für n Einträge):

Insertion Sort:	$c_1 * n^2$	(c_1 Konstante unabhängig von n)
Merge Sort:	$c_2 * n * \log_2 n$	(c_2 andere Konstante unabhängig von n)

Im Allgemeinen gilt $c_1 < c_2$.

Bei kleinem n ist Insertion Sort schneller als Merge Sort, aber ganz gleich um wieviel kleiner c_1 gegenüber c_2 ist, es wird einen Übergangspunkt geben für n, ab dem Sortieren durch Mischen schneller ist.

Bespiel: Rechner A 10^9 Instruktionen pro Sekunde, c_1 sei 2
Rechner B 10^7 Instruktionen pro Sekunden, c_2 sei 50
für $n = 10^6$ Einträge, Rechenzeit A = 2000 Sekunden, Rechenzeit B ~ 100 Sekunden

Auswahl eines Algorithmus

- Entwicklung von Alternativen (z.B. drei):
  - Analyse der Algorithmen (mathematisch, empirisch)
  - Begründete Auswahl
  - Implementierung
  - Test

  Regel: nur 1 einfachster Algorithmus o.k. ,wenn Algorithmus nur wenig genutzt oder wenn erste schnelle Lösung im Gesamtsystem gefordert

  Fehler 1: Missachtung der Leistungsmerkmale, schnellerer Algorithmus ist oft nicht komplizierter

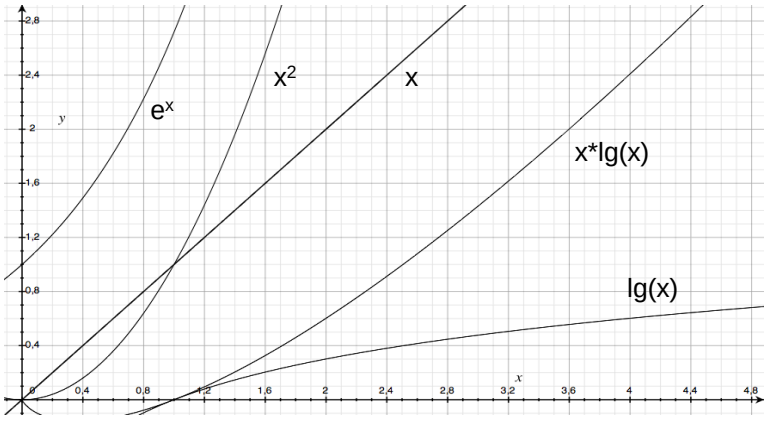
  Fehler 2: Leistungsmerkmale erhalten zu viel Aufmerksamkeit

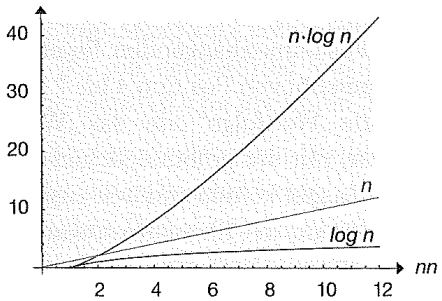
- Zur iterativen Verbesserung der Lösung (sukzessive Verfeinerung):
  - Finde innere Schleife
  - Prüfe jeden Befehl der inneren Schleife (z.B. keine Prozeduraufrufe)
  - Ersetze rekursiv durch iterativ
  - Letztlich implementiere in Assembler

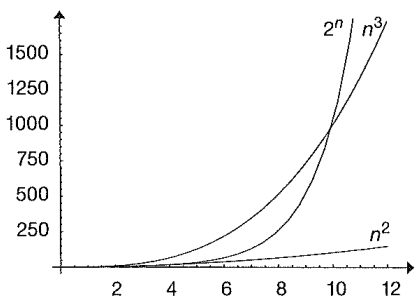
Laufzeitkomplexitätsklassen

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, weil *log* n im Verhältnis 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 Kapitel 7)
$O(n^3)$	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. *Backtracking-* oder *Exhaustionsalgorithmen* (siehe Kapitel 9)

Beispiel: Komplexitätsverhalten







Zwischenübung: Typische Wachstumsgesetze

Vergleich von Rechenzeiten:

Bestimmen Sie die Rechenzeiten für ein Problem mit $n=10^3$ für die in der Tabelle gelisteten Wachstumsgesetze

Verwendet wird ein Rechner mit 10^9 Instruktionen pro Sekunde, und ein Algorithmus mit 1000 Instruktionen.

	Rechenzeit [sec]
$\log_2 n$	
$\sqrt{n}$	
n	
$n \log_2 n$	
n^2	
n^3	
2^n	
$n!$	

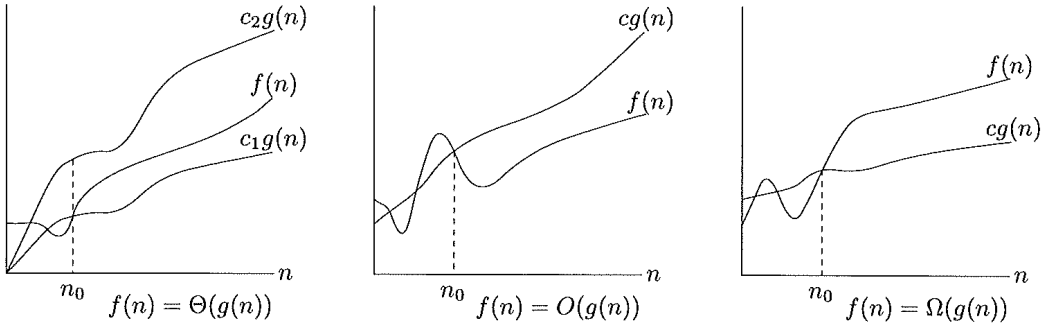
Zwischenübung: Typische Wachstumsgesetze

Bestimmen Sie für jede Funktion f(n) und für jede Zeit t den größten Wert von n für ein Problem, das innerhalb der Zeit t gelöst werden kann:

	1 Sekunde	1 Minute	1 Stunde	1 Tag	1 Monat	1 Jahr	1 Jahrhundert
$\log_2 n$							
$\sqrt{n}$							
n							
$n \log_2 n$							
n^2							
n^3							
2^n							
$n!$							

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

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



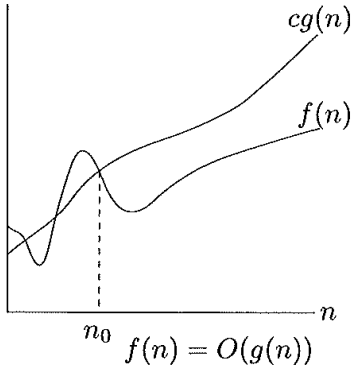
O Notation asymptotische Laufzeitkomplexität

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): es existieren positive Konstanten c und n_0, so dass

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



gesucht :

möglichst “kleine” und “einfache” Funktionen g(n)

mit möglichst geringen konstanten Faktoren c

als asymptotische obere Schranke für f(n)

Beispiel asymptotische Laufzeitkomplexität

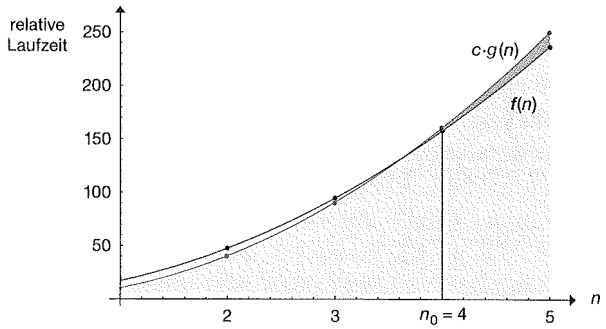
Ermittlung der asymptotischen Laufzeitkomplexität
des Algorithmus A1 aus Folie 8-22

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

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

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

Daraus ist ersichtlich, dass der Wert der Konstanten c gemäß Definition der O-Notation auf jeden Fall größer als 8,1 sein muss.
Für diese Funktion f gilt also schon für $n_0 = 4$, dass $f(n) \leq c \cdot g(n)$ mit $c = 10$ und $g(n) = n^2$ ist.



Somit gilt

$$f(n) = O(n^2)$$

oder anders ausgedrückt

$$f(n) \text{ ist von der Ordnung } (n^2)$$