Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München

Wiederholung letzte Vorlesung I

Annahmen im RAM-Modell

- nur sequentielle Ausführungen
  - das heißt nur 1 Prozessor, keine Parallelität
- alle Daten liegen direkt zugreifbar im Speicher ($\rightarrow$ RAM)
- jeder Speicherzugriff dauert gleich lang
  - das ist in Wahrheit nicht der Fall! (Speicher-Hierarchie)
- alle elementaren Verarbeitungsschritte benötigen eine Zeiteinheit
  - Wertzuweisung
  - Arithmetische Operationen: $+$, -, *, $/$, %, ceil, floor
  - Vergleichsoperationen: $<$, $>$, $!=$
  - Kontrollflußoperationen: **if**, **else**

Wiederholung letzte Vorlesung I

Landau-Symbol Θ

Landau-Symbol Θ

Sei $g : \mathbb{R} \to \mathbb{R}$ eine Funktion. Das Landau-Symbol $\Theta(g)$ ist definiert als die Menge

$$\Theta(g) := \{ f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

Für $f : \mathbb{R} \to \mathbb{R}$ mit $f \in \Theta(g)$ schreiben wir kurz: $f = \Theta(g)$.

- f ist also bis auf konstanten Faktor im wesentlichen "gleich" der Funktion g für $n \geq n_0$
- man sagt auch: g ist asymptotisch scharfe Schranke von f (von oben und unten)
- Kurznotation: $f = \Theta(g)$ oder $f(n) = \Theta(g(n))$

Annahmen

- nur se
  - d
- alle D
- jeder
  - d
- alle el
  Zeitei
  - V
  - A
  - V
  - Kontrollflußoperationen: **if**, **else**

17

Wiederholung letzte Vorlesung I

Landau-Symbol Θ

$$\Theta(g) = \{ f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

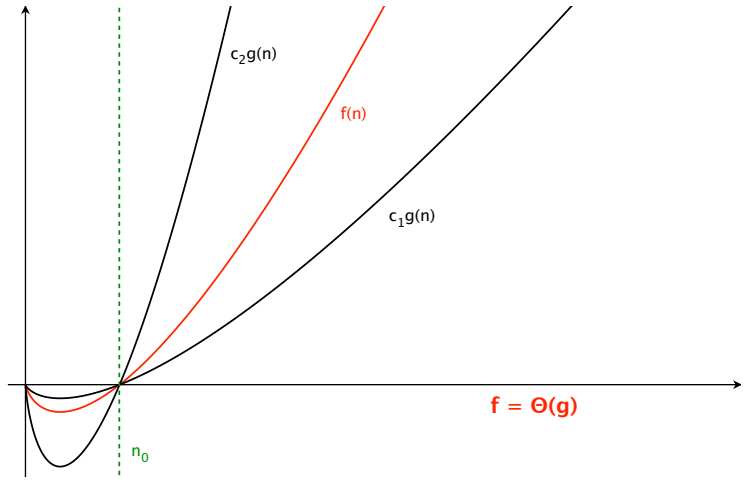
Landau-Syn

Landau-Sy

Sei $g : \mathbb{R}$ –
definiert al

$\Theta(g) :=$

Für $f : \mathbb{R}$ –



- f ist a
  der Fu
- man sagt auch: g ist asymptotisch scharfe Schranke von f (von oben und unten)
- Kurznotation: $f = \Theta(g)$ oder $f(n) = \Theta(g(n))$

18

Annahmen

- nur se
  - d
- alle D
- jeder
  - d
- alle el
  Zeitei
  - V
  - A
  - V
  - Kontrollflußoperationen: **if**, **else**

17

Wiederholung letzte Vorlesung I

Landau-Symbol Θ

$$\Theta(g) = \{f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Landau-Syn

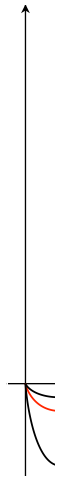
Landau-Sy

Sei $g : \mathbb{R}$ –
definiert al

$\Theta(g) :=$

Für $f : \mathbb{R}$ –

- f ist a der Fu
- man sagt auch: g ist a (von oben und unten)
- Kurznotation: $f = \Theta(g$



Annahmen

- nur se
  - d
- alle D
- jeder
  - d
- alle el Zeitei
  - V
  - A
  - V
- Kontrollflußoperationen: **if**, **else**

Landau-Symbol O

Landau-Symbol O

Sei $g : \mathbb{R} \to \mathbb{R}$ eine Funktion. Das Landau-Symbol $O(g)$ ist definiert als die Menge

$$O(g) := \{f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq f(n) \leq cg(n)\}$$

Für $f : \mathbb{R} \to \mathbb{R}$ mit $f \in O(g)$ schreiben wir kurz: $f = O(g)$.

- man sagt auch: g ist asymptotisch obere Schranke von f
- auch genannt: "gross-O-Notation"
- aus $f = \Theta(g)$ folgt automatisch $f = O(g)$

23

Wiederholung letzte Vorlesung I

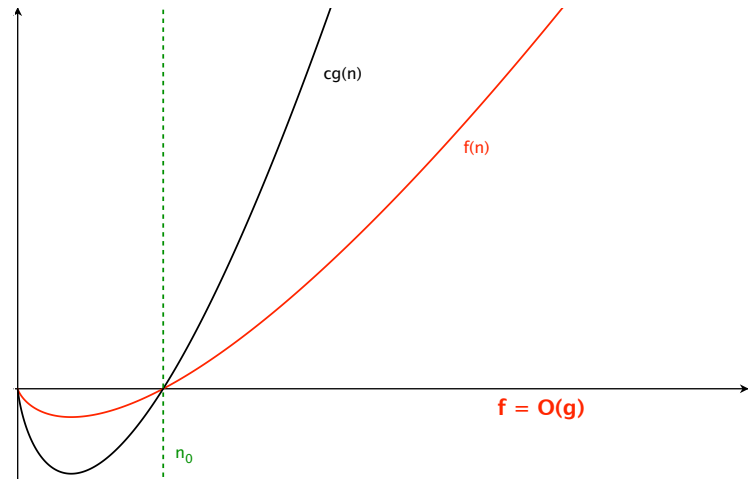
Landau-Symbol Θ

$$\Theta(g) = \{f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass}$$
$$\text{für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

Landau-Symbol O

Landau-Symbol O

$$O(g) = \{f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass}$$
$$\text{für alle } n \geq n_0 \text{ gilt: } 0 \leq f(n) \leq cg(n)\}$$



Wiederholung letzte Vorlesung I

Landau-Symbol Θ

$$\Theta(g) = \{ f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

Landau-Symbol O

Landau-Symbol O

$O(g) = \{ f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass}$

Beispiel: lineare Funktion

$$f(n) = 60n + 12$$

- **Behauptung:** $f(n) = O(n)$
  - tatsächlich:

$$0 \leq 60n + 12 \leq cn \quad \left| \frac{1}{n} \right.$$

$$0 \leq 60 + \frac{12}{n} \leq c$$

also z.B. $n_0 = 12$ und $c = 61$ ✓

- **Behauptung:** $f(n) = O(n^2)$
  - tatsächlich:

$$0 \leq 60n + 12 \leq cn^2 \iff 0 \leq \frac{60}{n} + \frac{12}{n^2} \leq c$$

also z.B. $n_0 = 1$ und $c = 73$ ✓

Wiederholung letzte Vorlesung II

Effizienz von Algorithmen I

O-Notation erlaubt Kategorisierung der Effizienz von Algorithmen

- $O(1)$: konstante Laufzeit
  - unabhängig von Problemgröße
  - *Beispiel:* Löschen von erstem Element in verketteter Liste
- $O(\log n)$: logarithmische Laufzeit
  - Laufzeit wächst langsamer als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. Kapitel 6)
  - *Beispiel:* Suchen in sortierter Liste (Binäre Suche, s. Kapitel 8)
- $O(n)$: lineare Laufzeit
  - Laufzeit wächst vergleichbar zur Problemgröße
  - jedes Eingabe-Element erfordert $O(1)$ Arbeit
  - *Beispiele:* Suchen in unsortierter Liste, Löschen von Element in sequentieller Liste

Wiederholung letzte Vorlesung II

Effizienz von Algorithmen II

- $O(n \log n)$: "loglinear" Laufzeit
  - Laufzeit wächst schneller als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. Kapitel 6)
  - *Beispiele:* Quicksort (s. Kapitel 6), FFT (s. Kapitel 10)
- $O(n^2)$: quadratische Laufzeit
  - typisch für Algorithmen, die Element paarweise kombinieren
  - *Beispiele:* Insertion Sort, Matrix-Vektor Multiplikation (s. Kapitel 10)
- $O(n^3)$: kubische Laufzeit
  - *Beispiel:* Matrix-Matrix Multiplikation
- $O(2^n)$: exponentielle Laufzeit
  - auch als "unlösbar" bezeichnet (intractable)
  - *Beispiel:* Traveling Salesman (kürzeste Route, so dass alle Städte exakt einmal besucht)

Effizienz vo

O-Notatio

- $O(1)$:
  - u
  - *E*
- $O(\log$
  - L
  - t
  - *E*
- $O(n)$:
  - L
  - jedes Eingabe-Element erfordert $O(1)$ Arbeit
  - *Beispiele:* Suchen in unsortierter Liste, Löschen von Element in sequentieller Liste

Wiederholung letzte Vorlesung II

Wachstumsraten illustriert

Annahme: Rechner mit 1GHz Taktfrequenz

- eine Operation benötigt 1ns (nano-Sekunde)
- n bezeichne Anzahl Operationen

n	$T(n) = \ln(n)$	$T(n) = n$	$T(n) = n\ln(n)$	$T(n) = n^2$	$T(n) = 2^n$	$T(n) = n!$
10	$0.003\mu s$	$0.01\mu s$	$0.033\mu s$	$0.1\mu s$	$1\mu s$	3.63ms
20	$0.004\mu s$	$0.02\mu s$	$0.086\mu s$	$0.4\mu s$	1ms	77.1 years
30	$0.005\mu s$	$0.03\mu s$	$0.147\mu s$	$0.9\mu s$	1s	$8.4 \cdot 10^{15}$years
40	$0.005\mu s$	$0.04\mu s$	$0.213\mu s$	$1.6\mu s$	18.3min	
50	$0.006\mu s$	$0.05\mu s$	$0.282\mu s$	$2.5\mu s$	13 days	
100	$0.007\mu s$	$0.1\mu s$	$0.644\mu s$	$10\mu s$	$4 \cdot 10^{13}$years	
1000	$0.010\mu s$	$1.0\mu s$	$9.966\mu s$	1ms		
10000	$0.013\mu s$	$10\mu s$	$130\mu s$	100ms		
100000	$0.017\mu s$	0.1ms	1.67ms	10s		
$1 \cdot 10^6$	$0.020\mu s$	1ms	19.93ms	16.7min		
$1 \cdot 10^7$	$0.023\mu s$	0.01s	0.23s	1.16 days		
$1 \cdot 10^8$	$0.027\mu s$	0.1s	2.66s	115.7 days		
$1 \cdot 10^9$	$0.030\mu s$	1s	29.9s	31.7 years		

Tabelle adaptiert von "The Algorithm Design Manual", S. Skiena, Springer

8

Effizienz vo

- $O(n$ lo
  - L
  - t
  - E
- $O(n^2)$
  - t
  - E
  K
- $O(n^3)$
  - E

Effizienz vo

O-Notatio

- $O(1)$:
  - u
  - E
- $O(\log$
  - L
  - t
  - E
- $O(n)$:
  - L
  - jedes Eingabe-Element erfordert $O(1)$ Arbeit
  - *Beispiele:* Suchen in unsortierter Liste, Löschen von Element in sequentieller Liste

- $O(2^n)$: exponentielle Laufzeit
  - auch als "unlösbar" bezeichnet (intractable)
  - *Beispiel:* Traveling Salesman (kürzeste Route, so dass alle Städte exakt einmal besucht)

28

27

Wiederholung letzte Vorlesung II

Wachstumsraten illustriert

Annahme: Rechner mit 1GHz Taktfrequenz

- eine Operation benötigt 1ns (nano-Sekunde)
- n bezeichne Anzahl Operationen

n	$T(n) = \ln$
10	$0.003\mu s$
20	$0.004\mu s$
30	$0.005\mu s$
40	$0.005\mu s$
50	$0.006\mu s$
100	$0.007\mu s$
1000	$0.010\mu s$
10000	$0.013\mu s$
100000	$0.017\mu s$
$1 \cdot 10^6$	$0.020\mu s$
$1 \cdot 10^7$	$0.023\mu s$
$1 \cdot 10^8$	$0.027\mu s$
$1 \cdot 10^9$	$0.030\mu s$

Tabelle adap

Effizienz vo

- $O(n \log$
  - L
  - t
  - E
- $O(n^2)$
  - t
  - E
- $O(n^3)$
  - E
- $O(2^n)$: exponentielle L
  - auch als "unlösbar"
  - *Beispiel:* Traveling S
    Städte exakt einmal

Effizienz vo

O-Notatio

- $O(1)$:
  - u
  - E
- $O(\log$
  - L
  - t
  - E
- $O(n)$:
  - L
  - jedes Eingabe-Element erfordert $O(1)$ Arbeit
  - *Beispiele:* Suchen in unsortierter Liste, Löschen von Element in sequentieller Liste

Rechenregel für O-Notation I

Addition in O-Notation

Seien A_1, A_2 zwei Algorithmen mit Laufzeiten

$$T_1(n) = O(f(n)), \qquad T_2(n) = O(g(n))$$

für zwei Funktionen $f, g : \mathbb{R} \to \mathbb{R}$. Dann hat der Algorithmus $A = A_1; A_2$ (sequentielle Ausführung von A_1, A_2) die Komplexität

$$T(n) = T_1(n) + T_2(n) = O\big(\max(f(n), g(n))\big)$$

Beispiel: Sei $A_1 = O(\log n)$, $A_2 = O(n^2)$. Dann ist für $A = A_1; A_2$

$$A = O\big(\max(\log n, n^2)\big) = O(n^2)$$

Wiederholung letzte Vorlesung II

Wachstumsraten illustriert

Annahme: Rechner mit 1GHz Taktfrequenz

- eine Operation benötigt 1ns (nano-Sekunde)
- n bezeichne Anzahl Operationen

Effizienz vo

- $O(n \log$
  - L
  - t
  - E
- $O(n^2)$
  - t
  - E
  - K
- $O(n^3)$
  - E
- $O(2^n)$: exponentielle L
  - auch als "unlösbar"
  - *Beispiel:* Traveling S
    Städte exakt einmal

n	$T(n) = \ln$
10	$0.003\mu s$
20	$0.004\mu s$
30	$0.005\mu s$
40	$0.005\mu s$
50	$0.006\mu s$
100	$0.007\mu s$
1000	$0.010\mu s$
10000	$0.013\mu s$
100000	$0.017\mu s$
$1 \cdot 10^6$	$0.020\mu s$
$1 \cdot 10^7$	$0.023\mu s$
$1 \cdot 10^8$	$0.027\mu s$
$1 \cdot 10^9$	$0.030\mu s$

Tabelle adap

Effizienz vo

O-Notatio

- $O(1)$:
  - u
  - E
- $O(\log$
  - L
  - t
  - E
- $O(n)$:
  - L
  - jedes Eingabe-Element erfordert $O(1)$ Arbeit
  - *Beispiele:* Suchen in unsortierter Liste, Löschen vo sequentieller Liste

Rechenregel für O-Notation I

Addition in O-Notation

Seien

für z
$A =$

Beisp

Komplexität der elementaren Bausteine

- Elementarer Verarbeitungsschritt:
  - $O(1)$
- Sequenz:
  - Addition in O-Notation
- Bedingter Verarbeitungsschritt:
  - Maximum von Komplexität von if und else Block, sowie
  - $O(1)$ für Auswertung der Bedingung
- Wiederholung:
  - Anzahl Wiederholungen multipliziert mit Komplexität Schleifenkörper, sowie
  - Anzahl Wiederholungen multipliziert mit $O(1)$ für Auswertung der Schleifenbedingung

Organisatorisches: Termine

- Wiederholungs-Vorlesung: Mittwoch, 5. Dezember 2012
  - Grundlagen: Kapitel 1 - 6
- keine Vorlesung: Donnerstag, 6. Dezember 2012 (Dies academicus)
- Probe-Klausur: Montag, 10. Dezember 2012
- Vorlesungs-Evaluation: Donnerstag, 13. Dezember 2012
- ab Mittwoch, 12. Dezember 2012: Fortgeschrittene Algorithmen und Datenstrukturen

Organisatorisches: eLearning

- Nachrichtenforum von Moodle:
  - Themen für jede Woche mit zusätzlichem Lernmaterial
  - Thema zum Ablauf des Übungsbetriebes
  - Thema für generelle Fragen und Antworten (FAQ)

  $\rightarrow$ bitte Möglichkeiten nutzen!

- Kummerkasten: anonymes Feedback zur Vorlesung/Übung
  - Anregungen, Beschwerden, Kommentare jederzeit willkommen!
  - Fragen können hier nicht beantwortet werden

- Ausführliche Gliederung auf Vorlesungswebseite
  - fortlaufend aktualisiert

Programm heute

Algorithmen-Entwurf

- Kein Patentrezept zum Entwurf von Algorithmen!
  - insbesondere Ableitung von Algorithmus aus Spezifikation nicht automatisierbar
- Programmieren ist kreative Tätigkeit
  - "The Art of Computer Programming" (D. Knuth)
- Unterstützung durch Algorithmen-Muster
  - auch Design Patterns genannt
  - "best practice"

Entwurfsprinzip: Verfeinerung

- intuitive Vorgehensweise: schrittweise Verfeinerung
- auch genannt: Top-Down Entwurf

- Schema:
  1. Lösungsidee mit groben Anweisungen
  2. Schrittweise Verfeinerung der Anweisungen in detailliertere / konkretere Anweisungen
  3. Zielprogramm in Programmiersprache (z.B. C)

Verfeinerung: Beispiel I

Beispiel-Problem: Tee-Zubereitung

- Lösungsidee mit groben Anweisungen:
  1. Wasser kochen
  2. Wasser in die Tasse
  3. Teebeutel in die Tasse
  4. Teebeutel herausnehmen

Verfeinerung: Beispiel I

Beispiel-Problem: Tee-Zubereitung

- Lösungsidee mit groben Anweisungen:
  1. Wasser kochen
  2. Wasser in die Tasse
  3. Teebeutel in die Tasse
  4. Teebeutel herausnehmen
- Verfeinerung:
  1. Wasser kochen
     1. Wasser in Wasserkocher füllen
     2. Wasserkocher anstecken
     3. Wasserkocher anschalten
     4. Warten bis Wasser kocht
     5. Wasserkocher ausschalten
  2. Wasser in die Tasse
     1. Wasserkocher abstecken
     2. Wasser in Tasse füllen
  3. ...

Verfeinerung: Beispiel II

Beispiel-Problem: bestimme Median von Zahlenfolge

Definition Median

Sei $(x_1, \ldots, x_n) \subset \mathbb{N}$ eine aufsteigend sortierte Folge natürlicher Zahlen mit $x_1 \leq x_2 \leq \ldots \leq x_n$. Der Median $\tilde{x}$ ist definiert als

$$\tilde{x} := x_{\lceil \frac{n}{2} \rceil}$$

Beispiele:

- Folge 3, 4, 7, 10, 11: Median ist 7
- Folge 21, 33, 47, 111: Median ist 33

(*Bemerkung:* manchmal ist der Median im Fall von n gerade auch als arithmetisches Mittel von $x_{\frac{n}{2}}$ und $x_{\frac{n}{2}+1}$ definiert)

Verfeinerung: Beispiel II

Beispiel-Problem: bestimme Median von Zahlenfolge

- Lösungsidee mit groben Anweisungen (Pseudocode):
  1. Eingabe Zahlenfolge X;
  2. Sortiere X;
  3. Bestimme Median *xt* von X;
  4. Gebe *xt* aus;

Verfeinerung: Beispiel II

Beispiel-Problem: bestimme Median von Zahlenfolge

- Lösungsidee mit groben Anweisungen (Pseudocode):
  1. Eingabe Zahlenfolge X;
  2. Sortiere X;
  3. Bestimme Median *xt* von X;
  4. Gebe *xt* aus;
- Verfeinerung (Pseudocode):
  1. X = EingabeFolge(); // Eingabe der Folge
  2. Sort(X); // rufe Sortiermethode auf
  3. xt = X[ceil(X.laenge / 2)]; // bestimme Median
  4. Output(xt); // gib xt aus

Verfeinerung: Beispiel II

Beispiel-Problem: bestimme Median von Zahlenfolge

- Lösungsidee mit groben Anweisungen (Pseudocode):
  1. Eingabe Zahlenfolge X;
  2. Sortiere X;
  3. Bestimme Median *xt* von X;
  4. Gebe *xt* aus;
- Verfeinerung (Pseudocode):
  1. X = EingabeFolge(); // Eingabe der Folge
  2. Sort(X); // rufe Sortiermethode auf
  3. xt = X[ceil(X.laenge / 2)]; // bestimme Median
  4. Output(xt); // gib xt aus
- Verfeinerung (C++):
  1. int X[] = {3, 4, 7, 10, 11}; // Folge fest vorgegeben
  2. int length = 5; // fixe Länge
  3. std::sort(X, X+5); // sortiere Folge
  4. int xt = X[ceil(length/ 2)]; // bestimme Median
  5. std::cout << "Median ist " << xt; // gib xt aus

Verfeinerung: Beispiel II

Beispiel-Problem: bestimme Median von Zahlenfolge

- Lösungsidee mit groben Anweisungen (Pseudocode):
  1. Eingabe Zahlenfolge X;
  2. Sortiere X;
  3. Bestimme Median *xt* von X;
  4. Gebe *xt* aus;
- Verfeinerung (Pseudocode):
  1. X = EingabeFolge(); // Eingabe der Folge
  2. Sort(X); // rufe Sortiermethode auf
  3. xt = X[ceil(X.laenge / 2)]; // bestimme Median
  4. Output(xt); // gib xt aus
- Verfeinerung (C++):
  1. int X[] = {3, 4, 7, 10, 11}; // Folge fest vorgegeben
  2. int length = 5; // fixe Länge
  3. std::sort(X, X+5); // sortiere Folge
  4. int xt = X[ceil(length/ 2)]; // bestimme Median
  5. std::cout << "Median ist " << xt; // gib xt aus
- ...

Verfeinerung

- keine festen Regeln für Verfeinerungs-Prozess
- passende Notation für Detailstufe verwenden (z.B. Pseudocode)
- jeder Verfeinerungsschritt enthält Entwurfs-Entscheidungen
  - bestimmt z.B. Art der Eingabe von Daten
  - bestimmt z.B. Laufzeitverhalten des Programms (Größe, Geschwindigkeit)

Entwurfsprinzip: Algorithmen-Muster

- Idee: allgemeines Algorithmen-Muster entwickelt für bestimmte Problemklasse, dann Anpassung an konkretes Problem
- auch genannt: Design Pattern oder "best practice" Strategie

Entwurfsprinzip: Algorithmen-Muster

- Idee: allgemeines Algorithmen-Muster entwickelt für bestimmte Problemklasse, dann Anpassung an konkretes Problem
- auch genannt: Design Pattern oder "best practice" Strategie

- Beispiel Problemklasse: finden einer optimalen Lösung bezüglich Kosten in großem Lösungsraum
  - konkret z.B. kürzester Weg zwischen zwei Städten

Entwurfsprinzip: Algorithmen-Muster

- Idee: allgemeines Algorithmen-Muster entwickelt für bestimmte Problemklasse, dann Anpassung an konkretes Problem
- auch genannt: Design Pattern oder "best practice" Strategie

- Beispiel Problemklasse: finden einer optimalen Lösung bezüglich Kosten in großem Lösungsraum
  - konkret z.B. kürzester Weg zwischen zwei Städten

- Verfahrensweise Algorithmen-Muster:
  - Beschreibung des Lösungsverfahrens an möglichst einfachem Beispiel
  - Transfer auf konkrete Problemstellung

Programm heute

Entwurfsprinzip: Divide and Conquer

Definition: Divide and Conquer (nach G. Saake)

Divide and Conquer ist die rekursive Rückführung eines zu lösenden Problems auf ein identisches Problem mit kleinerer Eingabemenge.

- Divide and Conquer: zu deutsch "Teile und herrsche"
- Prinzip:
  - teile große Aufgabe in mehrere kleine Teilaufgaben
  - rufe denselben Algorithmus rekursiv auf den Teilaufgaben auf

Muster: Divide and Conquer

Divide and Conquer als Algorithmen-Muster:

1. Teile gegebene Aufgabe in mehrere getrennte Teilaufgaben
   - löse Teilaufgaben einzeln
   - setze Lösung der Gesamtaufgabe aus Teillösungen zusammen
2. Wende dieselbe Technik auf jede Teilaufgabe an, dann auf deren Teilaufgaben etc., bis die Teilaufgabe so klein ist, dass Lösung explizit berechnet werden kann
3. Jede Teilaufgabe sollte von derselben Art sein wie die Gesamtaufgabe, so dass der gleiche Algorithmus rekursiv aufgerufen werden kann

Muster: Divide and Conquer

Divide and Conquer-Muster als Pseudocode:

Input: Aufgabe A
DivideAndConquer(A):
  if (A klein) {
    löse A explizit;
  }
  else {
    teile A in Teilaufgaben $A_1, \ldots, A_n$ auf;
    DivideAndConquer(A_1);
    ...
    DivideAndConquer(A_n);
    setze Lösung für A zusammen aus Lösungen für $A_1, \ldots, A_n$
}

Divide and Conquer: MergeSort

Sei $A = \{a_1, \ldots, a_n\}$ Feld mit n natürlichen Zahlen $a_i \in \mathbb{N}$.

Aufgabe: sortiere A in aufsteigender Reihenfolge.

- Lösung mit Divide and Conquer-Muster: Merge Sort
- Idee:
  - Divide: teile A auf in zwei gleich große Teilfelder
  - Rekursion: rufe Merge Sort rekursiv für die zwei Teilfelder auf
  - Conquer: setze die Teilfelder zusammen (merge bzw. mischen)
- Wann ist Teilfolge "klein", d.h. wann löst man explizit?
  $\rightarrow$ Teilfolge mit nur einem Element $\rightarrow$ sortiert!

MergeSort: Algorithmus

Input: zu sortierendes Feld A
Output: sortiertes Feld

MergeSort(A):
```
if (A ein-elementig) { // Teilfeld ist klein
    return A;
}
else {
    halbiere A in A_1 und A_2; // Divide
    A_1 = MergeSort(A_1); // Rekursion
    A_2 = MergeSort(A_2); // Rekursion
    return Merge(A_1, A_2); // Conquer
}
```

→ "komplizierter" Teil ist in **Merge**!

MergeSort: Merge Funktion

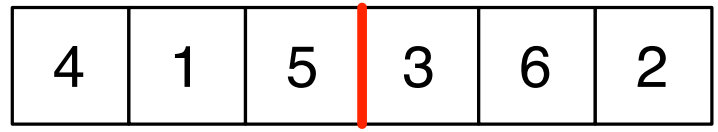
Input: zu sortierende Felder A_1, A_2
Output: sortiertes Feld B

Merge(A_1, A_2):
  B = leeres Feld;
  while (A_1 und A_2 nicht leer) {
    entferne das kleinere der Anfangselemente von A_1 bzw. A_2;
    hänge dieses Element an B an;
  }
  hänge das verbliebene, nicht-leere Feld A_1 oder A_2 an B an;
  return B;

$\rightarrow$ tatsächliche Implementation muß Entfernen/Anhängen effizient lösen!

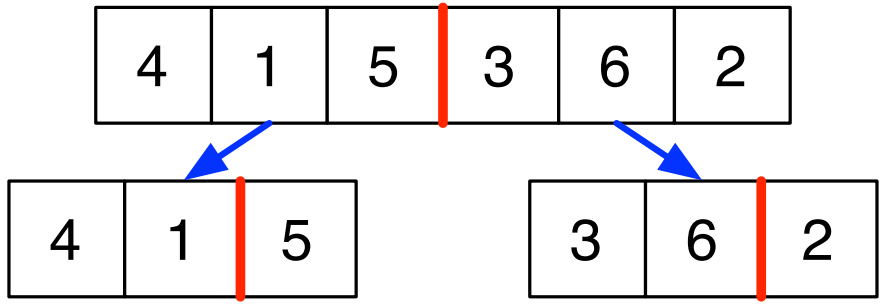
MergeSort: Beispiel

Divide



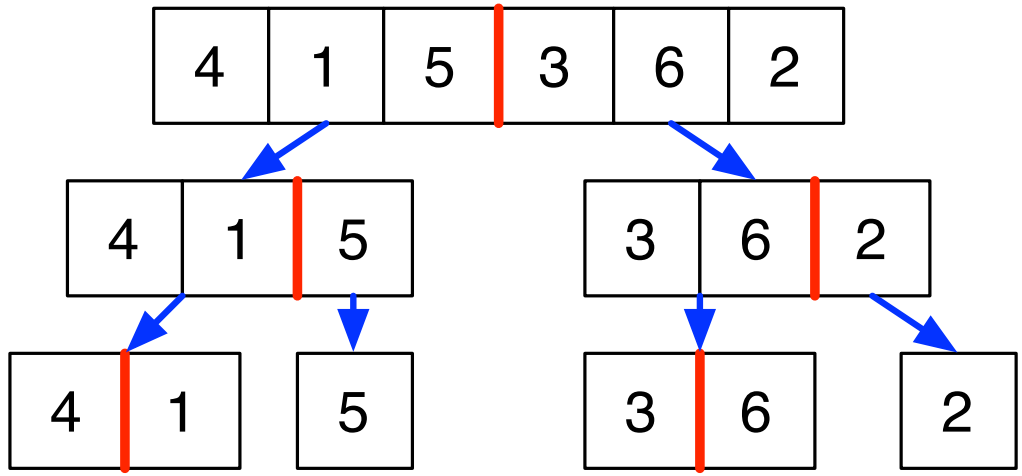
MergeSort: Beispiel

Divide



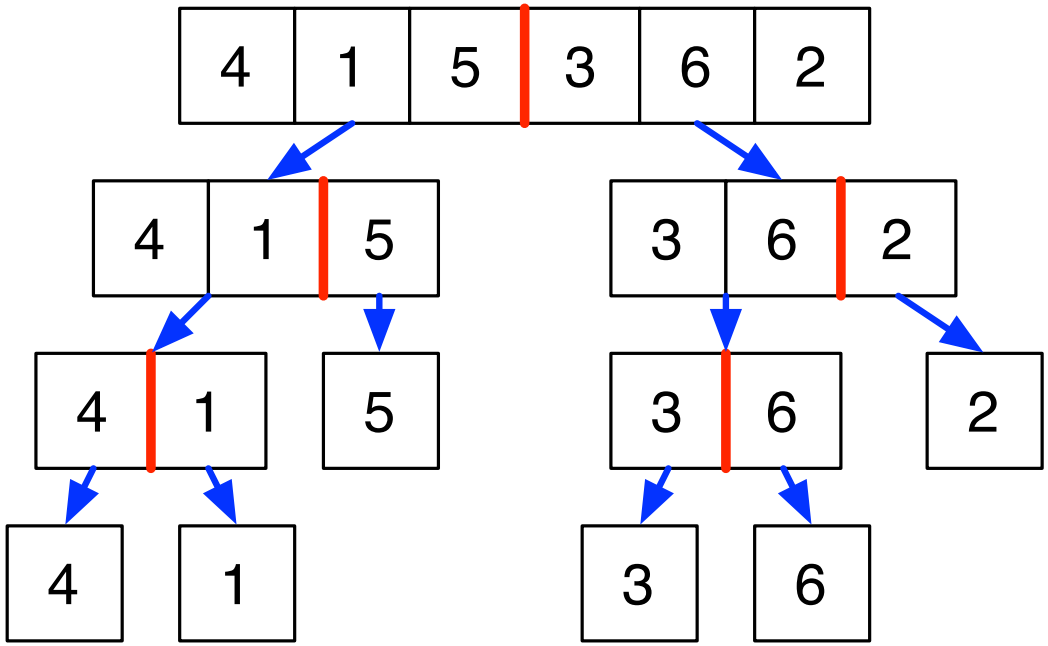
MergeSort: Beispiel

Divide

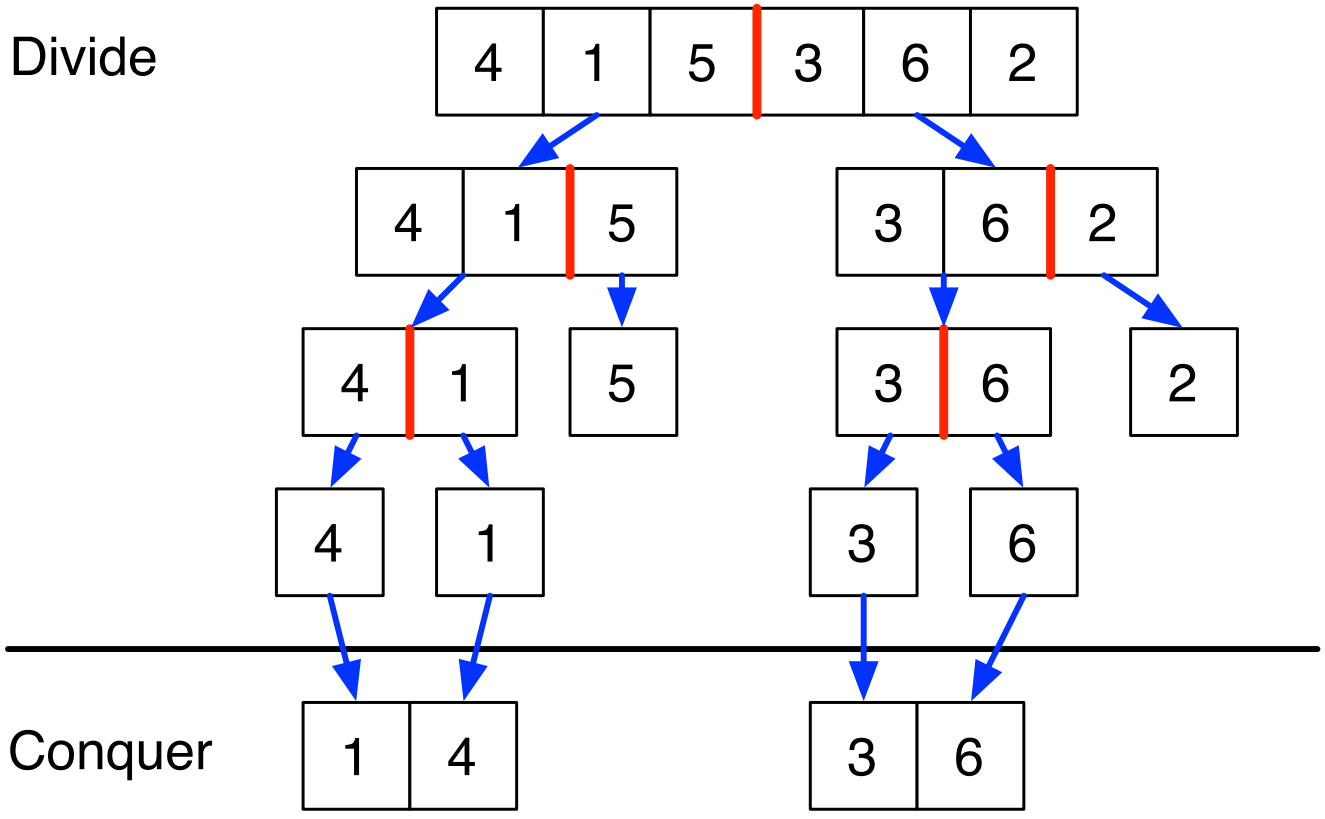


MergeSort: Beispiel

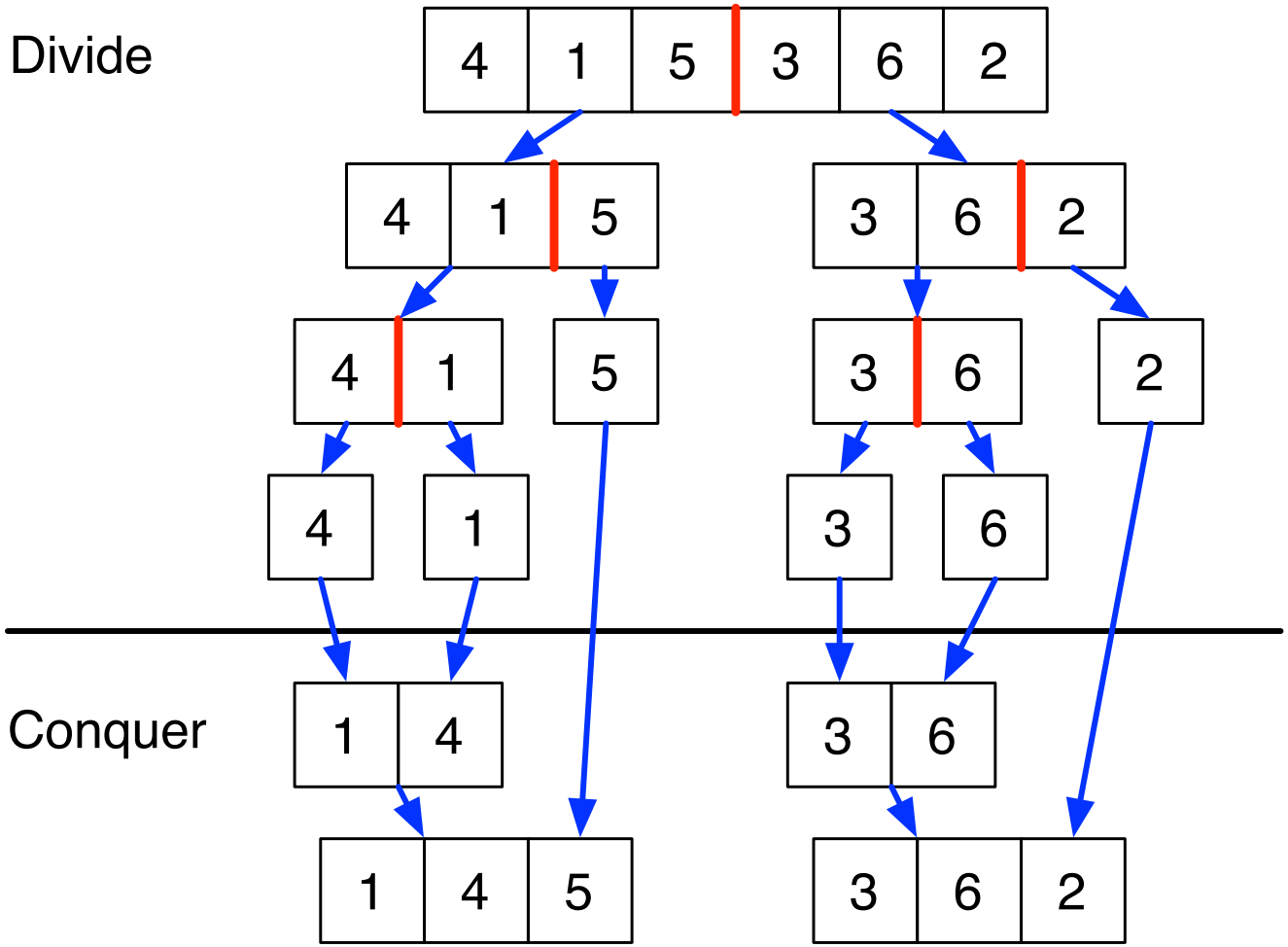
Divide



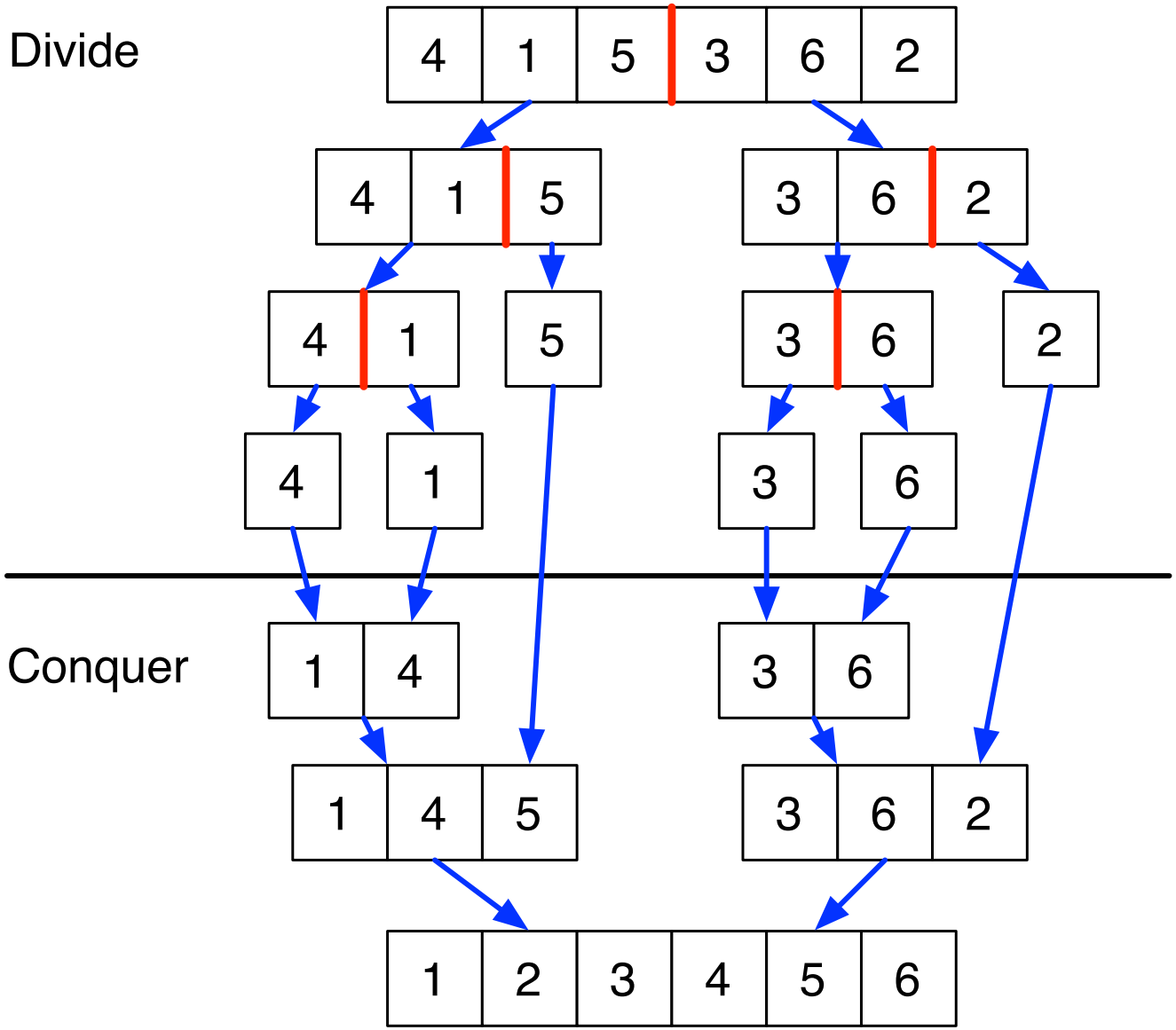
MergeSort: Beispiel



MergeSort: Beispiel



MergeSort: Beispiel



MergeSort: Eigenschaften

- Merge Sort benötigt zusätzlichen Speicher in Funktion **Merge**
  - insgesamt n zusätzliche Elemente falls A Länge n
- best und worst case sind identisch
- die meiste Arbeit steckt in **Merge**
  - wie viel genau?

MergeSort: Komplexität Merge Funktion

- $|A| = n$
- Sei $|A_1| = n_1$, $|A_2| = n_2$
  $\rightarrow n_1 + n_2 \leq n$
- Annahme:
  - entfernen ist $O(1)$
  - anhängen ist $O(1)$
- Anzahl **while** Durchläufe:
  $\min(n_1, n_2)$
- anschließend
  $\max(n_1, n_2) - \min(n_1, n_2)$
  Anhänge-Operationen
  $\rightarrow$ **Merge** ist $O(n_1 + n_2)$
- also ist **Merge** auch $O(n)$!

Input: zu sortierende Felder A_1, A_2
Output: sortiertes Feld B

Merge(A_1, A_2):
  B = leeres Feld;
  while (A_1 und A_2 nicht leer) {
    entferne das kleinere der Anfangs-
    elemente von A_1 bzw. A_2;
    hänge dieses Element an B an;
  }
  hänge das verbliebene, nicht-leere
  Feld A_1 oder A_2 an B an;
  return B;

MergeSort: Komplexität I

- Laufzeit von Merge Sort
  $T(n) = ?$
- Zeilen 1-3: "Problem klein"
  $O(1)$
- Zeile 5: Divide
  $O(1)$
- Zeile 6 und 7: Rekursion
  jeweils $T(n/2)$
- Zeile 8: Conquer
  $O(n)$

Input: zu sortierendes Feld A
Output: sortiertes Feld

MergeSort(A):

```
1    if (A ein-elementig) {
2       return A;
3    }
4    else {
5       halbiere A in A_1 und A_2;
6       A_1 = MergeSort(A_1);
7       A_2 = MergeSort(A_2);
8       return Merge(A_1, A_2);
9    }
```

MergeSort: Komplexität I

- Laufzeit von Merge Sort
  $T(n) = ?$
- Zeilen 1-3: "Problem klein"
  $O(1)$
- Zeile 5: Divide
  $O(1)$
- Zeile 6 und 7: Rekursion
  jeweils $T(n/2)$
- Zeile 8: Conquer
  $O(n)$

Input: zu sortierendes Feld A
Output: sortiertes Feld

MergeSort(A):

```
1    if (A ein-elementig) {
2       return A;
3    }
4    else {
5       halbiere A in A_1 und A_2;
6       A_1 = MergeSort(A_1);
7       A_2 = MergeSort(A_2);
8       return Merge(A_1, A_2);
9    }
```

$$\rightarrow T(n) = \begin{cases} O(1) & \text{für } n = 1 \\ 2T(n/2) + O(n) & \text{für } n > 1 \end{cases}$$

MergeSort: Komplexität II

$$T(n) = \begin{cases} O(1) & \text{für } n = 1 \\ 2T(n/2) + O(n) & \text{für } n > 1 \end{cases}$$

liefert durch Auflösen der Rekursionsgleichung

$$T(n) = \Theta(n \log n)$$

Wie kann man das zeigen?

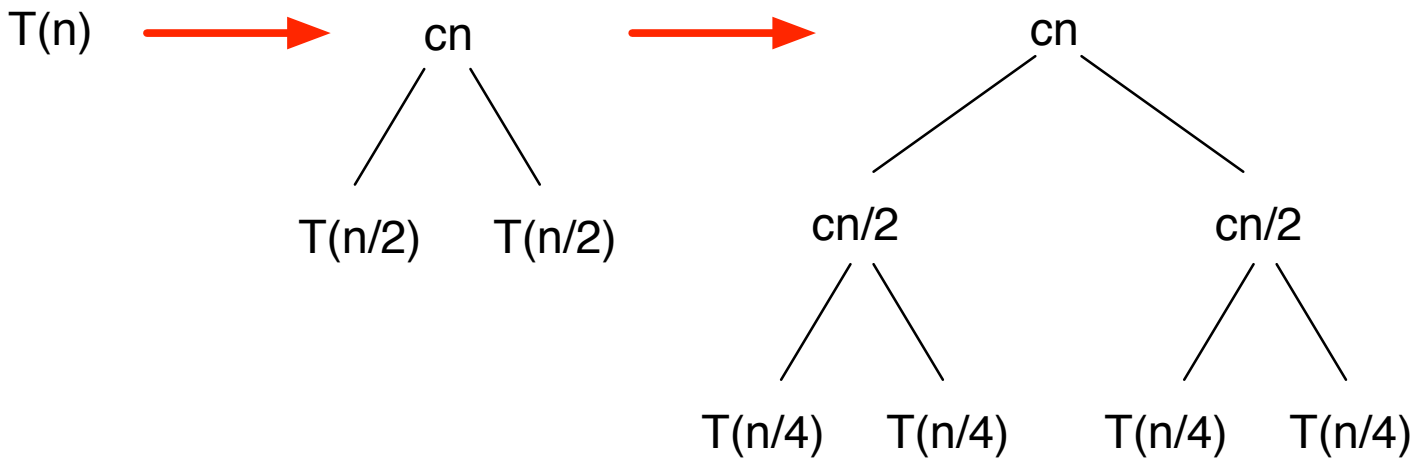
- Master-Theorem (s. Cormen, Kapitel 4.5)
- intuitiver: Rekursions-Baum

  hierzu Annahme: c passende Konstante, so daß

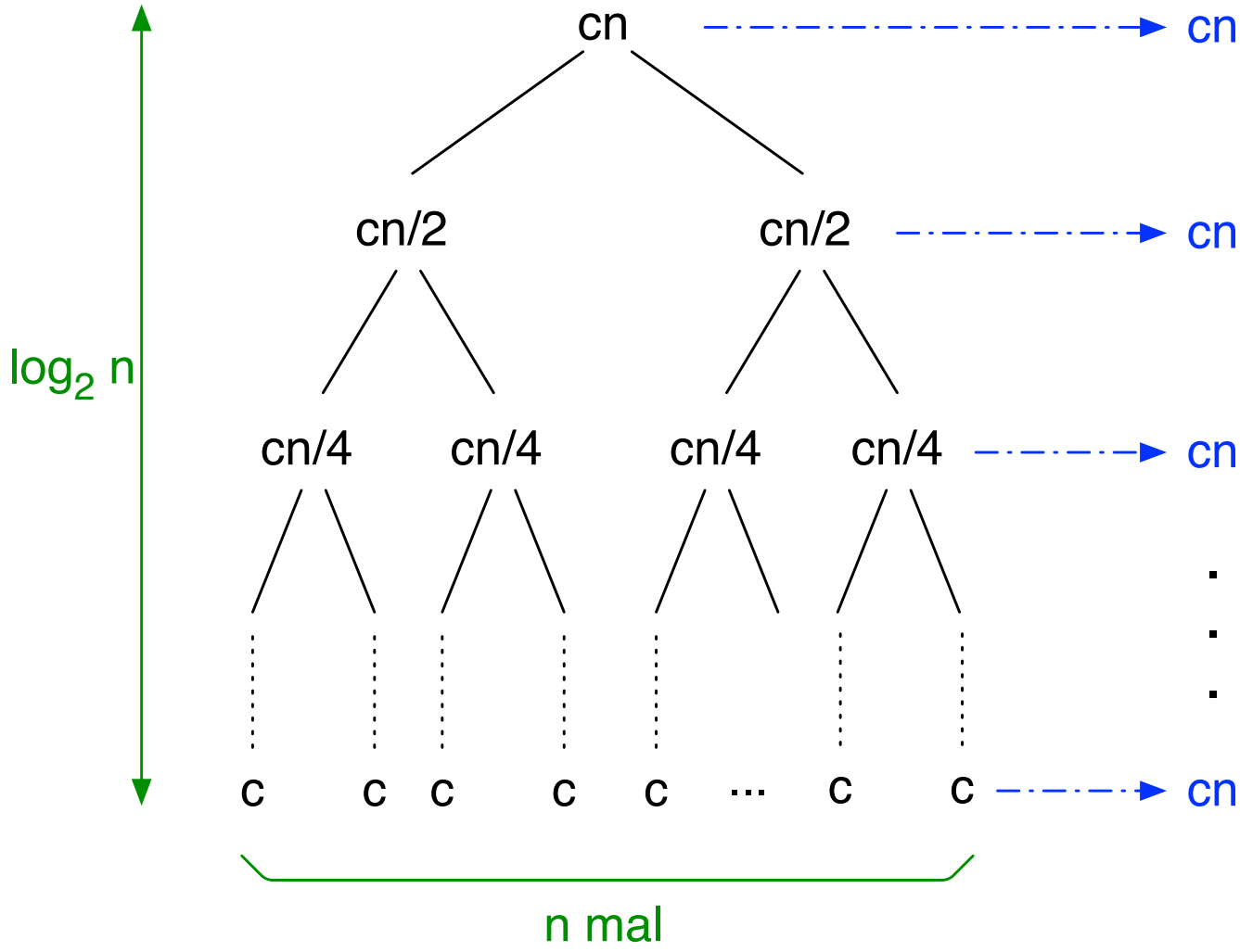
$$T(n) = \begin{cases} c & \text{für } n = 1 \\ 2T(n/2) + cn & \text{für } n > 1 \end{cases}$$

  und sei n Zweier-Potenz, d.h. $n = 2^k$ für $k \in \mathbb{N}$

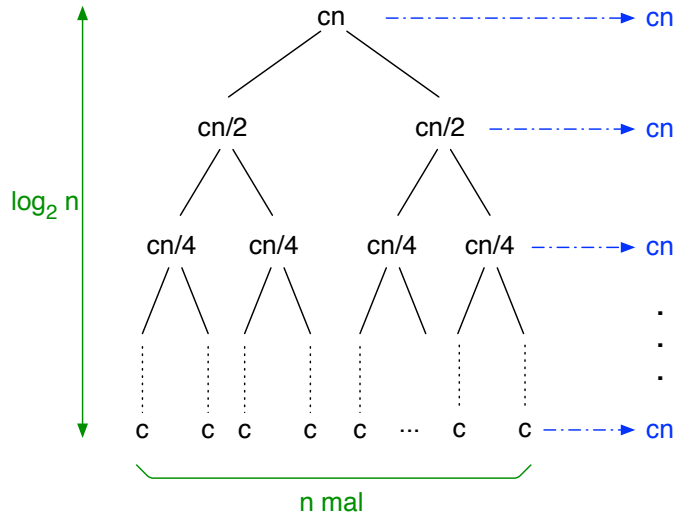
MergeSort: Rekursions-Baum



MergeSort: Rekursions-Baum



MergeSort: Komplexität III



- Baum hat Höhe $\log_2 n$, also $\log_2 n + 1$ Ebenen
- pro Ebene cn Kosten

$\rightarrow$ Gesamtkosten $cn(\log_2 n) = cn \log_2 n + cn$

$\longrightarrow T(n) = \Theta(n \log n)$ ✓

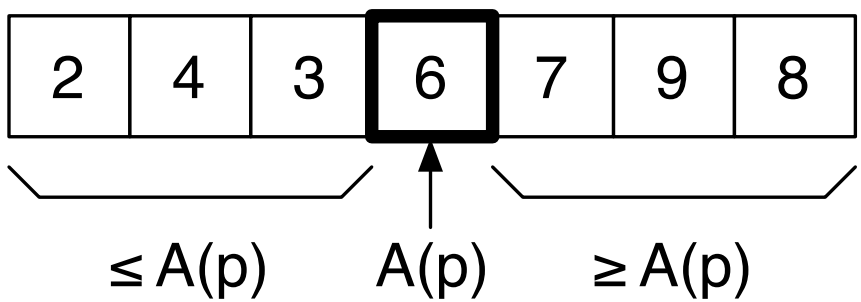
Divide and Conquer: QuickSort

QuickSort:

- Muster: Divide and Conquer
- in-place (ohne extra Speicherbedarf)

Hauptidee:

- teile Feld A mittels Pivot-Element $A(p)$ in zwei Teile
- links vom Pivot-Element $A(p)$ sind alle Elemente kleiner als $A(p)$
- rechts vom Pivot-Element $A(p)$ sind alle Elemente größer als $A(p)$



QuickSort: Algorithmus

Input: zu sortierendes Feld A
Indices untere und obere Grenze u, o

QuickSort:(A, u, o)
if $(o > u)$ {
bestimme Index p des Pivot-Elements; // ist beliebig
$pn =$ **Partition**(A, u, o, p); // Divide
QuickSort$(A, u, pn - 1)$; // Rekursion
QuickSort$(A, pn + 1, o)$; // Rekursion
}

$\rightarrow$ "Intelligenz" in **Partition**

QuickSort: Funktion Partition I

Input: zu zerlegendes Feld A,
Indices von unterer und oberer Grenze u, o,
Position p des Pivot-Elements

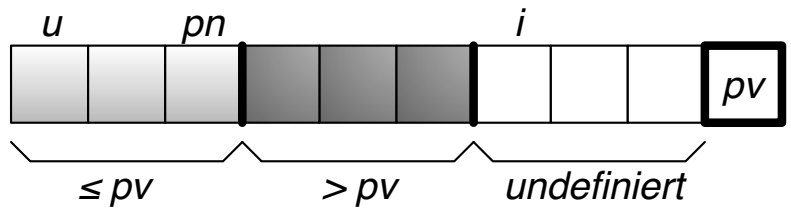
Output: neue Position pn des Pivot-Elements

Partition(A, u, o, p)

```
pn = u;
pv = A(p);
tausche A(p) und A(o); // Pivot-Element nach rechts
for i = u to o − 1 {
  if (A(i) ≤ pv) {
    tausche A(pn) und A(i);
    pn = pn + 1;
  }
}
tausche A(o) und A(pn); // Pivot-Element zurück
return pn;
```

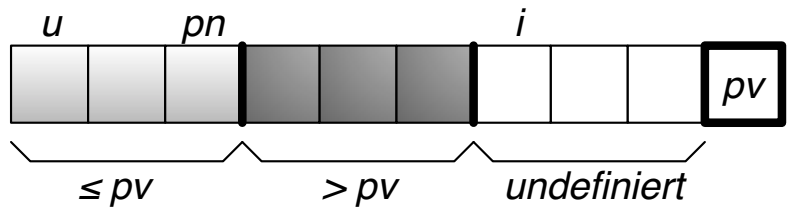
QuickSort: Funktion Partition II

Partitions-Prinzip:

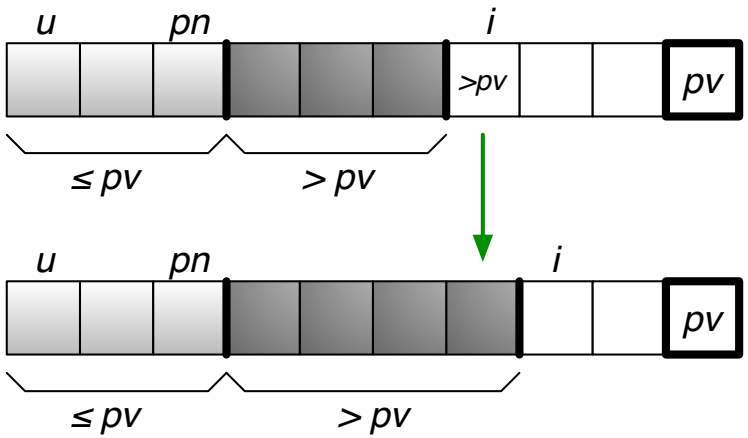


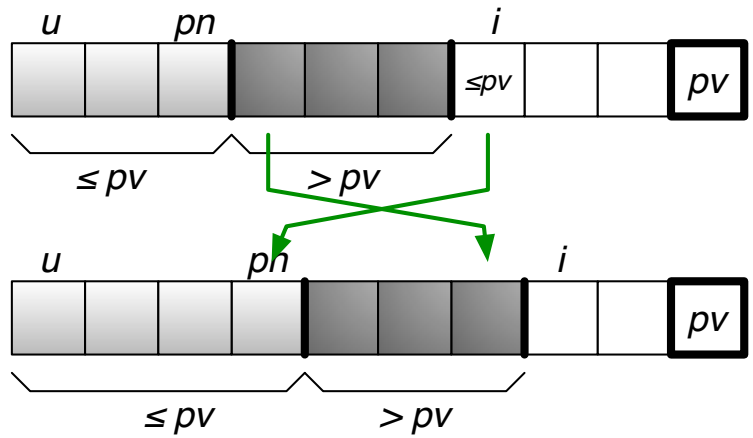
QuickSort: Funktion Partition II

Partitions-Prinzip:

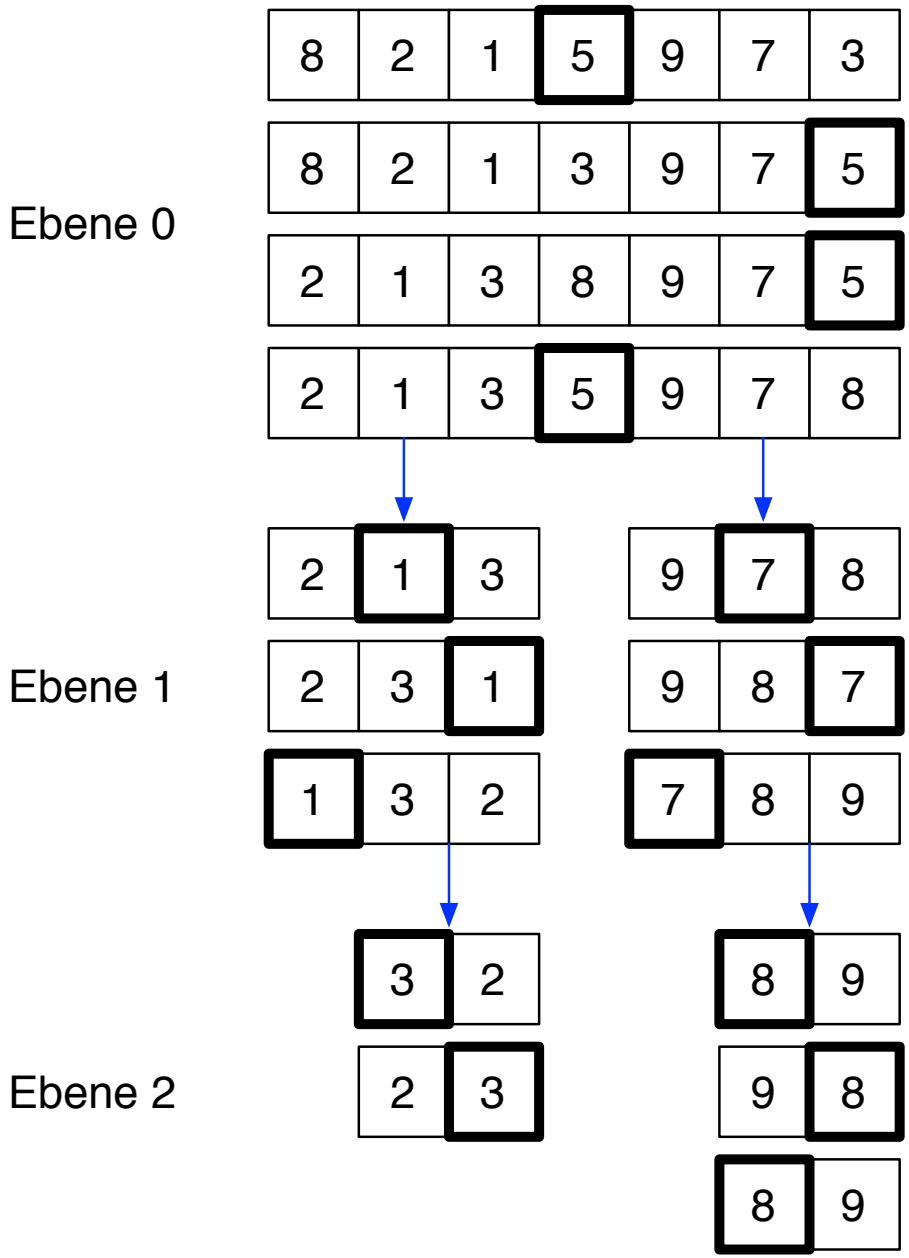


Austausch-Schritt:





QuickSort: Beispiel



QuickSort: Komplexität

Komplexität von QuickSort:

- worst case: $\Theta(n^2)$
  - tritt ein wenn Pivot-Element ständig unbalanciert ist (d.h. ein Teilfeld leer)
  - insbesondere wenn Feld bereits sortiert
- best case: $O(n \log n)$
  - wenn Pivot-Element Folge laufend halbiert
- im Durchschnitt: $O(n \log n)$
  - Beweis aufwendig
- typische Verbesserung: randomisierte Wahl von Pivot-Element
  - Komplexität dann immer noch $O(n \log n)$

Bei Interesse an Beweis/Herleitung: s. Cormen, Kapitel 7.2 und 7.4

Vergleich Sortier-Algorithmen

- Insertion Sort
  - in-place
  - Komplexität $O(n^2)$, best case: $O(n)$
- Selection Sort (Übung)
  - in-place
  - Komplexität $O(n^2)$
- MergeSort
  - benötigt zusätzlichen Speicher
  - Komplexität $O(n \log n)$
- QuickSort
  - in-place
  - Komplexität im Mittel $O(n \log n)$, worst case: $O(n^2)$

Zusammenfassung