

Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2015

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München



Programm heute

- ① Einführung
- ② Grundlagen von Algorithmen
- ③ Grundlagen von Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
- ⑤ Grundlagen der Effizienz von Algorithmen
- ⑥ Grundlagen des Algorithmen-Entwurfs
Entwurfsprinzipien
Divide and Conquer

3

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 Lösungs idee mit groben Anweisungen
 - ② Schrittweise Verfeinerung der Anweisungen in detailliertere / konkretere Anweisungen
 - ③ Zielprogramm in Programmiersprache (z.B. C++)

4

5

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 ...

6

Verfeinerung: Beispiel II

Beispiel-Problem: bestimme Median von Zahlenfolge

Definition Median

Sei $(x_1, \dots, x_n) \subset \mathbb{N}$ eine aufsteigend sortierte Folge natürlicher Zahlen mit $x_1 \leq x_2 \leq \dots \leq x_n$. 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)

7

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 $X = \text{EingabeFolge}();$ // Eingabe der Folge
 - 2 $\text{Sort}(X);$ // rufe Sortiermethode auf
 - 3 $xt = X[\text{ceil}(X.\text{laenge} / 2)];$ // bestimme Median
 - 4 $\text{Output}(xt);$ // gib xt aus
- Verfeinerung (C++):
 - 1 `std::vector<int> X{3, 4, 7, 10, 11};` // Folge fest vorgegeben
 - 2 `int length = X.size();` // fixe Länge
 - 3 `std::sort(X);` // sortiere Folge
 - 4 `int xt = X[ceil(length/ 2)];` // bestimme Median
 - 5 `std::cout << "Median ist " << xt;` // gib xt aus
- ...

8

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)

9

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

10

Programm heute

- ① Einführung
- ② Grundlagen von Algorithmen
- ③ Grundlagen von Datenstrukturen
- ④ Grundlagen der Korrektheit von Algorithmen
- ⑤ Grundlagen der Effizienz von Algorithmen
- ⑥ Grundlagen des Algorithmen-Entwurfs
 - Entwurfsprinzipien
 - Divide and Conquer

11

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

12

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

13

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  $A_1, \dots, A_n$  auf;
  DivideAndConquer( $A_1$ );
  ...
  DivideAndConquer( $A_n$ );
  setze Lösung für  $A$  zusammen aus Lösungen für  $A_1, \dots, A_n$ 
}
```

14

Divide and Conquer: MergeSort

Sei $A = \{a_1, \dots, a_n\}$ 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?
→ Teilfolge mit nur [einem](#) Element → sortiert!

15

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
   $A_1 = \mathbf{MergeSort}(A_1)$ ; // Rekursion
   $A_2 = \mathbf{MergeSort}(A_2)$ ; // Rekursion
  return Merge( $A_1, A_2$ ); // Conquer
}
```

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

16

MergeSort: Merge Funktion

Input: zu sortierende Felder A_1, A_2

Output: sortiertes Feld B

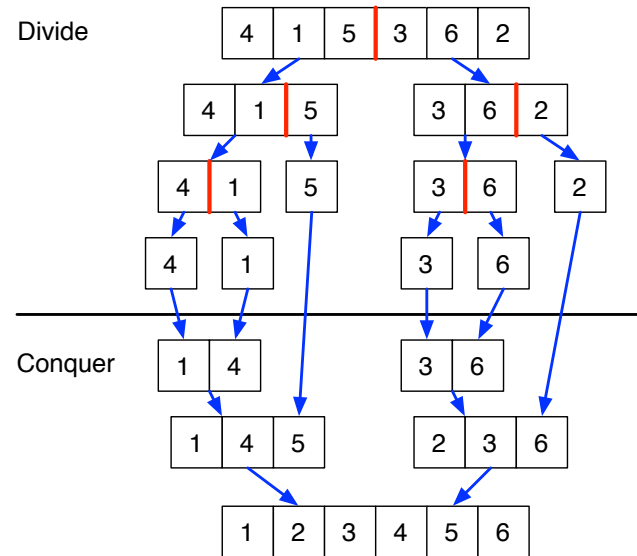
Merge(A_1, A_2):

```
 $B =$  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;
}
häng das verbliebene, nicht-leere Feld  $A_1$  oder  $A_2$  an  $B$  an;
return  $B$ ;
```

→ tatsächliche Implementation muß Entfernen/Anhängen effizient lösen!

17

MergeSort: Beispiel



18

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?

19

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 (A1 und A2 nicht leer) {
    entferne das kleinere der Anfangs-
    elemente von A1 bzw. A2;
    hänge dieses Element an B an;
}
hänge das verbliebene, nicht-leere
Feld A1 oder A2 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 A1 und A2;
6    A1 = MergeSort(A1);
7    A2 = MergeSort(A2);
8    return Merge(A1, A2);
9  }
    
```

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

20

21

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

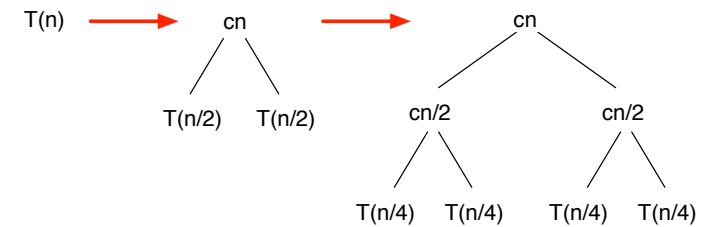
hierzu Annahme: c passende Konstante, so daß

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

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

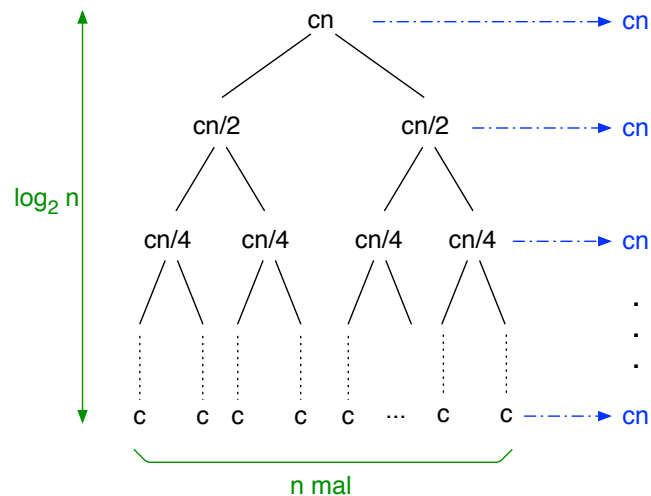
22

MergeSort: Rekursions-Baum

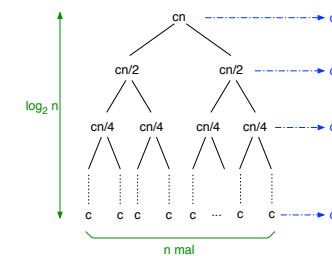


23

MergeSort: Rekursions-Baum



MergeSort: Komplexität III




- Baum hat Höhe $\log_2 n$, also $\log_2 n + 1$ Ebenen
- pro Ebene cn Kosten
 → Gesamtkosten $cn(\log_2 n + 1) = cn \log_2 n + cn$

$$\rightarrow T(n) = \Theta(n \log n) \quad \checkmark$$

24

25

MergeSort: Illustration

 <http://www.sorting-algorithms.com/merge-sort>

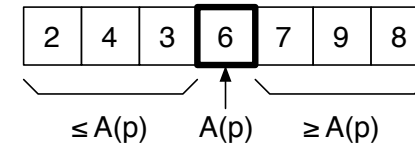
Divide and Conquer: QuickSort

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 $A(p)$



26

27

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  
     $pn = \text{Partition}(A, u, o, p)$ ; // Divide  
    QuickSort( $A, u, pn - 1$ ); // Rekursion  
    QuickSort( $A, pn + 1, o$ ); // Rekursion  
  }
```

→ "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

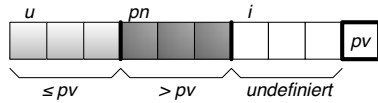
```
Partition( $A, u, o, p$ )  
   $pn = u$ ;  
   $pv = A(p)$ ;  
  tausche  $A(p)$  und  $A(o)$ ; // Pivot-Element nach rechts  
  for  $i = u$  to  $o - 1$  {  
    if ( $A(i) \leq pv$ ) {  
      tausche  $A(pn)$  und  $A(i)$ ;  
       $pn = pn + 1$ ;  
    }  
  }  
  tausche  $A(o)$  und  $A(pn)$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

28

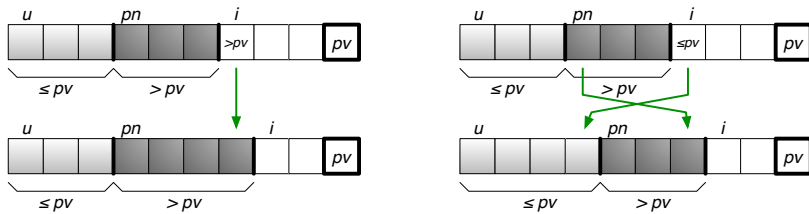
29

QuickSort: Funktion Partition II

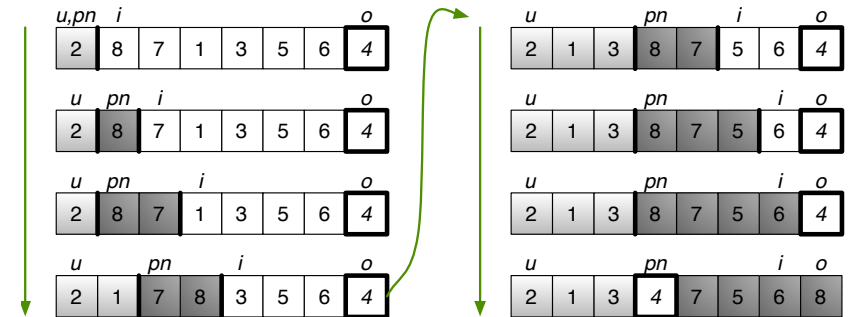
Partitions-Prinzip:



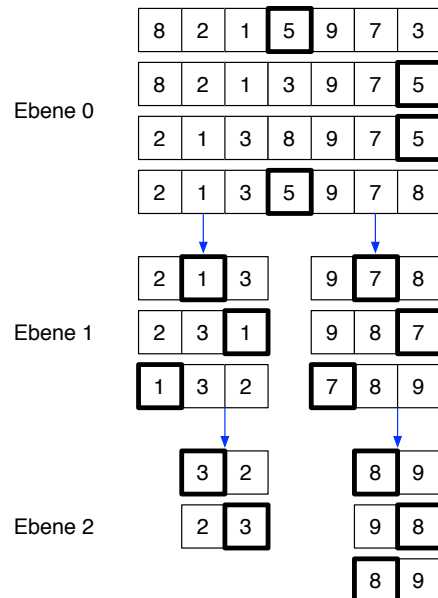
Austausch-Schritt:



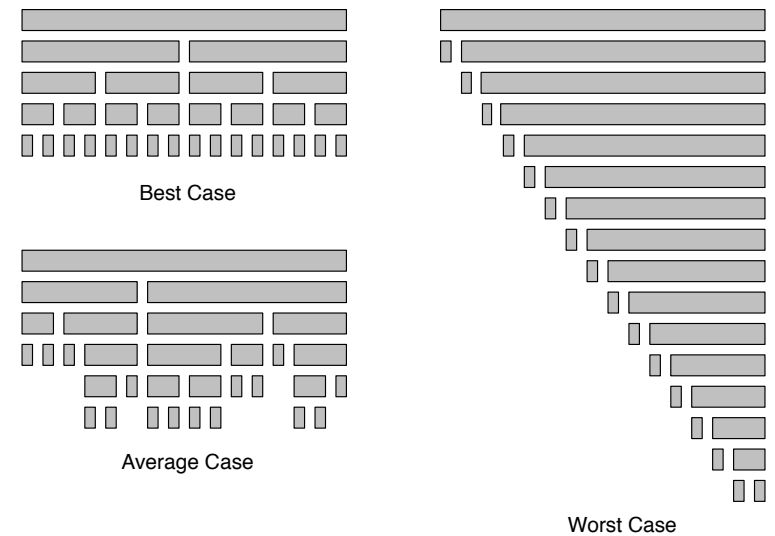
Partition: Beispiel



QuickSort: Komplettes Beispiel



Quicksort: Illustration best/worst case



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

34

QuickSort: Illustration

 <http://www.sorting-algorithms.com/quick-sort>

35

Sortier-Algorithmen Zusammenfassung

- 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: $O(n^2)$

36

Zusammenfassung

- 1 Einführung
- 2 Grundlagen von Algorithmen
- 3 Grundlagen von Datenstrukturen
- 4 Grundlagen der Korrektheit von Algorithmen
- 5 Grundlagen der Effizienz von Algorithmen
- 6 Grundlagen des Algorithmen-Entwurfs
 - Entwurfsprinzipien
 - Divide and Conquer

37