Übung Algorithmen I
22.4.15

Christoph Striecks
Christoph.Striecks@kit.edu

(Mit Folien von Julian Arz, Timo Bingmann und Sebastian Schlag.)

Roadmap

- Organisatorisches, Übungsblätter
- Effizienz von Algorithmen, O-Kalkül
- Korrektheit von Algorithmen
- Teile-und-Herrsche-Paradigma

Organisatorisches zur Übung

- Zusammen mit Christian Staudt (`Christian.Staudt@kit.edu`)
- Üblicherweise mittwochs, 14.45-15.30, Ausnahme: nächste Woche am 29.4.15, 14.00-15.30

Übungsblätter

M	D	M	D	F	S	S
V		V/Ü				
V		V/Ü		A		

- Ausgabe in der Regel nach der Übung (Ü)
- Abgabe (A) bis 12.45 Uhr freitags der darauf folgenden Woche, Briefkasten im Untergeschoss von Gebäude 50.34

Übungsblätter – wichtig!

- Nummer des Tutoriums groß auf erste Seite schreiben
- Handschriftliche Abgabe der Übungsblätter zu zweit
- Partner müssen im gleichen Tutorium sein
- Beschriftung der Übungsblätter:
  - Namen, Matrikelnummern, Nummer des Übungsblatts, Name des Tutors, Nummer des Tutoriums

Bonuspunkte für die Klausur

- 25% der Übungspunkte $\rightarrow$ **1** Bonuspunkt
- 50% der Übungspunkte $\rightarrow$ **2** Bonuspunkte
- 75% der Übungspunkte $\rightarrow$ **3** Bonuspunkte

Fragen, Diskussionen, Anregungen, ...

- Bevorzugtes Medium: ILIAS-Forum unter https://ilias.studium.kit.edu/goto_produktiv_crs_427875.html (Fragen zur Vorlesung, Übung, Tutorien, etc.)
- Feedbackkasten unter https://crypto.iti.kit.edu/algo-sose15 (Mit Bitte um konstruktive Beiträge.)

Buch zur Vorlesung

- Deutsches Buch zur Vorlesung auf der Webseite verlinkt
- Username, Passwort bei mir erfragen

Algorithmen

Effizienz von Algorithmen

Effizienz von Algorithmen

- Algorithmen können sich in ihrer Effizienz stark unterscheiden (auch wenn sie als Software auf unterschiedlicher Hardware laufen)
- Beispiel: Sortieren von n Zahlen (aus [CLRS04])

Sortieren durch Einfügen	Sortieren durch Mischen
$c_1 \cdot n^2$	$c_2 \cdot n \log n$

- Überlicherweise gilt $c_1 < c_2$

Effizienz von Algorithmen

- Setze $n = 10^6$, $c_1 = 2$ und $c_2 = 50$

Sortieren durch Einfügen	Sortieren durch Mischen
$2 \cdot (10^6)^2$ Anw.	$50 \cdot 10^6 \log 10^6$ Anw.
Rechner mit 2 GHz	Rechner mit 500 MHz
$\frac{2 \cdot (10^6)^2 \text{ Anw.}}{2 \cdot 10^9 \text{ Anw. pro sec}} = 1000 \text{ sec}$	$\frac{50 \cdot 10^6 \log 10^6 \text{ Anw.}}{500 \cdot 10^6 \text{ Anw. pro sec}} = 0.6 \text{ sec}$

- Mit $n = 10^7$:

Sortieren durch Einfügen	Sortieren durch Mischen
≈ 27.8 h	7 sec

Generelles Beispiel

- Laufzeit mit 10^9 Anw. pro Sekunde unter Eingabegröße n

n	$1000 \log_2 n$	$500n$	$100n \log_2 n$	$10n^2$	n^3	2^n
10	3.3 μs	5 μs	3.3 μs	1 μs	1 μs	1 μs
20	4.3 μs	10 μs	8.6 μs	4 μs	8 μs	1.05 ms
50	5.6 μs	25 μs	28.2 μs	25 μs	125 μs	13 h
100	6.6 μs	50 μs	66.4 μs	100 μs	1 ms	10^{13} a
250	8.0 μs	125 μs	199 μs	625 μs	15 ms	—
500	9.0 μs	250 μs	448 μs	2.5 ms	125 ms	
10^3	10.0 μs	500 μs	1 ms	10 ms	1 s	
10^4	13.3 μs	5 ms	13.3 ms	1 s	16 min	
10^5	16.6 μs	50 ms	166 ms	100 s	11.6 d	
10^6	20 μs	500 ms	2 s	2.7 h	31.7 a	
10^7	23 μs	5 s	23 s	11.6 d	$10^{4.5}$ a	
10^8	27 μs	50 s	4.4 min	3.2 a	$10^{7.5}$ a	

Eingabegröße und Laufzeit

- Eingabegröße:
  - hängt vom betrachteten Problem ab
  - Sortieren: Anzahl der Datensätze
  - Multiplikation: Anzahl der Bits (Darstellung der Zahlen)
  - Graph: Anzahl der Knoten und Kanten
- Laufzeit:
  - Anzahl der ausgeführten Grundoperationen ("Schritte")
  - gängige Annahme: jede Zeile im Pseudocode hat konstanten Zeitaufwand
  - Aufsummieren der Kosten jeder ausgeführten Zeile ergibt im wesentlichen Laufzeit

Genauer: (asymptotische) Laufzeit

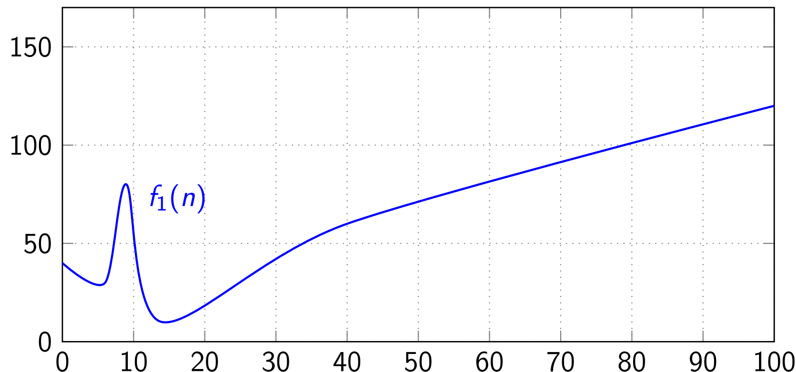
- Bestimmung im günstigsten, mittleren und schlechtesten Fall möglich
- üblich ist Bestimmung im schlechtesten Fall:
  - Laufzeit ist obere Schranke einer bel. Eingabe
  - häufiges Auftreten
- asymptotisch: mit der Größe der Eingabe im Limes

(Asymptotische) O-Notation

$$\mathrm{O}(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$

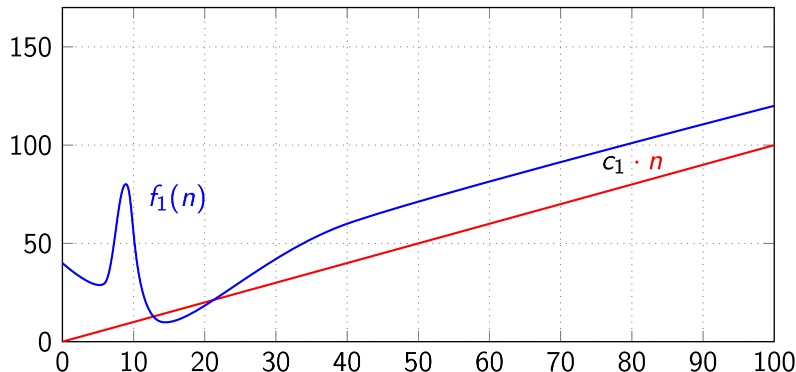
O-Notation (Intuition)

$$O(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$



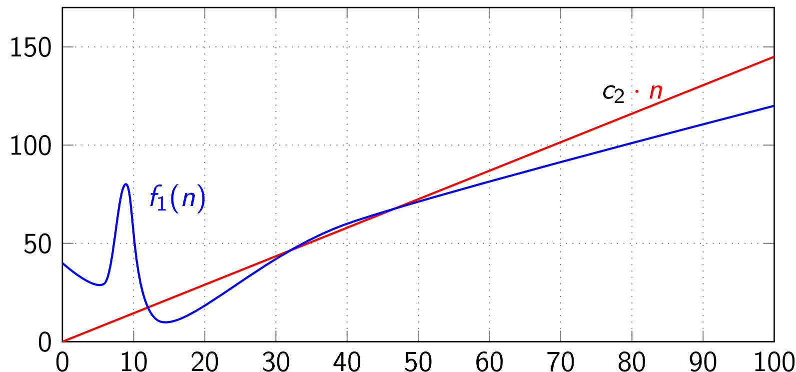
O-Notation (Intuition)

$$\mathrm{O}(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \\ \text{sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$



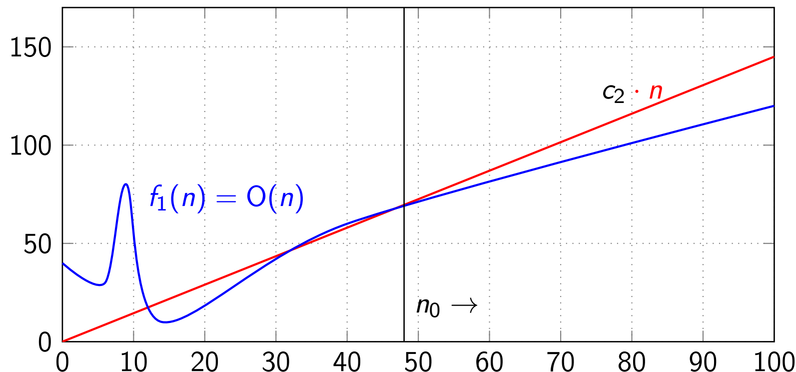
O-Notation (Intuition)

$$\mathrm{O}(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$



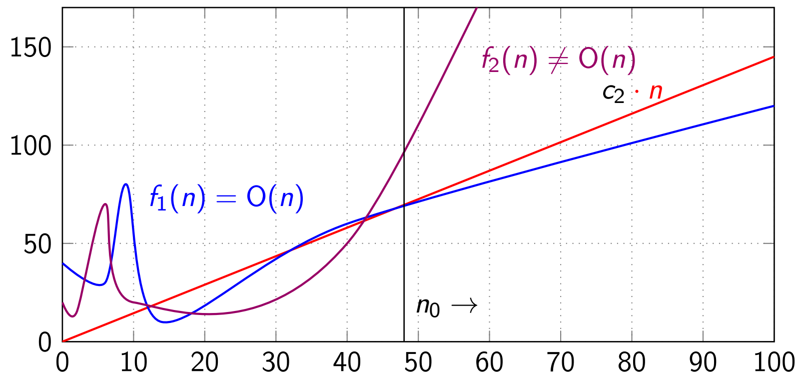
O-Notation (Intuition)

$$\mathrm{O}(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$



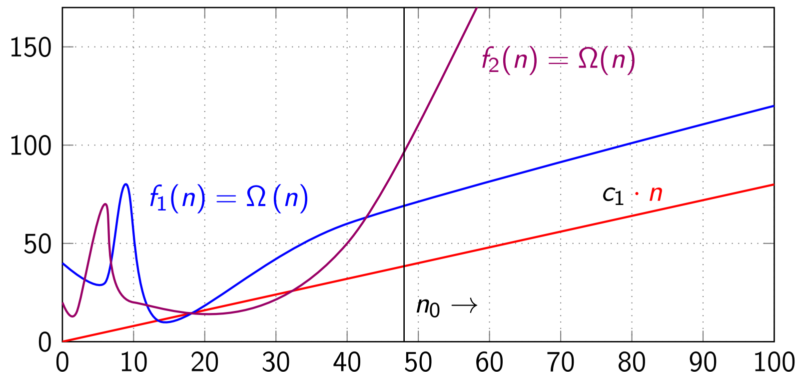
O-Notation (Intuition)

$$\mathrm{O}(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$



O-Notation (Intuition)

$$\mathrm{O}(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$

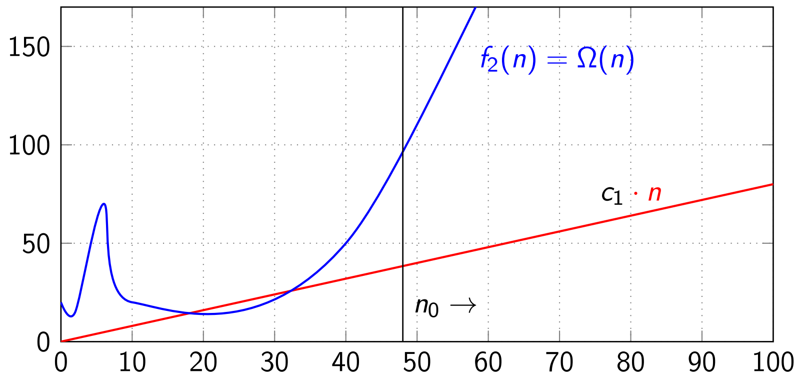


(Asymptotische) Ω-Notation (Intuition)

$$\Omega(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \\ \text{sodass } 0 \leq c \cdot g(n) \leq f(n) \text{ für alle } n \geq n_0\}$$

Ω-Notation (Intuition)

$$\Omega(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq c \cdot g(n) \leq f(n) \text{ für alle } n \geq n_0\}$$



Asymptotische Notationen

$$\mathrm{O}(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$$

$$\Theta(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c_1,\ c_2 \text{ und } n_0, \text{ sodass } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ für alle } n \geq n_0\}$$

$$\Omega(g(n)) = \{f(n) : \text{ es exist. pos. Konstanten } c \text{ und } n_0, \text{ sodass } 0 \leq c \cdot g(n) \leq f(n) \text{ für alle } n \geq n_0\}$$

Und weitere (siehe VL): $\mathrm{o}(g(n)), \omega(g(n))$

Basis des Logarithmus

- $O(\log n)$, aber zu welcher Basis?
- Üblicherweise zur Basis 2 in der Informatik
- Innerhalb der O-Notation “meistens egal”
- Denn:
$$O(\log_b n) = O(\frac{\log_a n}{\log_a b}) = O(\log_a n),$$
für beliebige a, b ($\log_a b$ ist unabhängig von n)

Korrektheit von Algorithmen

Invarianten

Schleifeninvarianten (Folien von J. Arz, T. Bingmann, S. Schlag)

- nützliches Tool für Korrektheitsbeweise
- Schleifeninvariante muss 3 Bedingungen erfüllen:

1. **Initialisierung:** Invariante gilt vor erster Iteration
2. **Fortsetzung:** Invariante gilt vor Iteration i
   $\Rightarrow$ Invariante gilt vor Iteration $i + 1$
3. **Terminierung:** Abbruchbed. erfüllt $\wedge$ Invariante gilt
   $\Rightarrow$ richtiges Ergebnis / Nachbedingung erfüllt

Invarianten

Schleifeninvarianten

1. **Initialisierung:** Invariante gilt vor erster Iteration
2. **Fortsetzung:** Invariante gilt vor Iteration i
   $\Rightarrow$ Invariante gilt vor Iteration $i + 1$
3. **Terminierung:** Abbruchbed. erfüllt $\wedge$ Invariante gilt
   $\Rightarrow$ richtiges Ergebnis / Nachbedingung erfüllt

- Wenn 1. und 2. erfüllt, dann ist Invariante wahr vor jeder Iteration der Schleife (Ähnlichkeit zur mathematischen Induktion, IA und IS)
- 3. Eigenschaft unterscheidet sich zur math. Induktion, da Schleife “abbricht”

Invarianten

Zusicherungen und Invarianten

- Vorbedingungen
- (Schleifen-)Invarianten
- Nachbedingungen

Function power$(a : \mathbb{R};\ n_0 : \mathbb{N}) : \mathbb{R}$
    assert $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$
    $p{=}a : \mathbb{R};\quad r{=}1 : \mathbb{R};\quad n{=}n_0 : \mathbb{N}$
    while $n > 0$ **do**
        invariant $p^n r = a^{n_0}$
        if n is odd **then** $n--$; $r{:=}\ r \cdot p$
        else $(n, p){:=}\ (n/2, p \cdot p)$
    assert $r = a^{n_0}$
    return r

Invarianten

Idee der Anwendung

- finde Schleifeninvariante
- zeige Schleifeninvariante
- Schleifeninvariante und sonstiges Wissen
  $\Rightarrow$ Korrektheit des Algorithmus

Function $\max(A : \textbf{Array } [0..n-1] \textbf{ of } \mathbb{R})$:
    assert $A.\text{size}() > 0$ // Vorbedingung
    i=0 : $\mathbb{N}_{\geq 0}$
    j=1 : $\mathbb{N}_{\geq 0}$
    while $j < n$ **do**
        if $A[j] > A[i]$ **then** $i := j$
        ++j
    assert $i = \text{argmax}_{\ell < n} A[\ell]$ // Nachbedingung
    return i

Invarianten

Beispiel I

i=0 : $\mathbb{N}_{\geq 0}$
j=1 : $\mathbb{N}_{\geq 0}$
while $j < n$ **do**
  invariant $i = \text{argmax}_{\ell < j} A[\ell]$ // Invariante
  if $A[j] > A[i]$ **then** $i := j$
  ++j

Beweis:

- $j = 1$: klar
- $j \to j + 1$: es gilt $i = \text{argmax}_{\ell < j+1} A[\ell]$, also $A[i] = \max_{\ell < j+1} A[\ell]$
  1. Fall 1: $A[j+1] > A[i] = \max_{\ell < j+1} A[\ell] \to$ update $i \to$ Beh.
  2. Fall 2: $A[j+1] \leq A[i] = \max_{\ell < j+1} A[\ell] \to$ kein update $\to$ Beh.

Invarianten

Beispiel I

```
Function max(A : Array [0..n − 1] of ℝ) :
    assert A.size() > 0                         // Vorbedingung
    i=0 : ℕ≥0
    j=1 : ℕ≥0
    while j < n do
        invariant i = argmax_{ℓ<j} A[ℓ]          // Invariante
        if A[j] > A[i] then i := j
        ++j
    assert i = argmax_{ℓ<n} A[ℓ]                 // Nachbedingung
    return i
```

- nach Beenden der Schleife gilt: $j = n$ und $i = \operatorname{argmax}_{\ell<j} A[\ell]$
- $\Rightarrow$ Nachbedingung gilt: $i = \operatorname{argmax}_{\ell<n} A[\ell]$
- $\Rightarrow$ Algorithmus liefert korrektes Ergebnis

Invarianten

Beispiel III

Ziel: Nach Terminierung enthält b die binäre Repräsentation von n

```
Function convertToBinary(n : ℕ) :
    b := Array < 0, ..., 0 >    // binary representation of n
    t := n
    k := −1
    while t > 0 do
        k := k + 1
        b[k] := t mod 2
        t := t div 2
    assert ∑_{i=0} b[i] × 2^i = n            // Nachbedingung
    return b
```

Invarianten

Beispiel III

Wieder drei Schritte zu tun:

1. Invar. vor Beginn der Schleife wahr
2. Invar. gilt im k-ten $\Rightarrow$ Invar. gilt im $(k+1)$-ten Schritt
3. Nach Terminierung, Invar. wahr $\Rightarrow$ Korrektheit des Algorithmus

$$\text{Invariante: } n = t \cdot 2^{k+1} + m$$

mit m die Zahl, die durch $b[0, \ldots, k]$ repräsentiert wird

Invarianten

Beispiel III - Anfang

m die Zahl, die durch $b[0,\ldots,k]$ repräsentiert wird

Function convertToBinary$(n : \mathbb{N})$:

- $b :=$ **Array** $< 0,\ldots,0 >$ // binary representation of n
- $t := n$
- $k := -1$
- **while** $t > 0$ **do**
  - **invariant** $n = t \cdot 2^{k+1} + m$
  - $k := k + 1$
  - $b[k] := t \bmod 2$
  - $t := t \text{ div } 2$
- **assert** $\sum_{i=0} b[i] \times 2^i = n$ // Nachbedingung
- **return** b

Invarianten

Beispiel III - IS

m die Zahl, die durch $b[0, \dots, k]$ repräsentiert wird

while $t > 0$ **do**

  invariant $n = t \cdot 2^{k+1} + m$

  $k := k + 1$

  $b[k] := t \bmod 2$

  $t := t \text{ div } 2$

Beweis $(k \to k+1)$**:**

- es gelte vor der Iteration $n = t \cdot 2^{k+1} + m$
- Fall 1: t gerade
  1. $k' := k + 1$
  2. $b[k'] = t \bmod 2 = 0$, m unverändert
  3. $t' := t/2$
- nach der Iteration:

$$t/2 \cdot 2^{k+2} + m = t \cdot 2^{k+1} + m = n$$

Invarianten

Beispiel III - IS

m die Zahl, die durch $b[0, \ldots, k]$ repräsentiert wird

while $t > 0$ **do**
  invariant $n = t \cdot 2^{k+1} + m$
  $k := k + 1$
  $b[k] := t \bmod 2$
  $t := t \operatorname{div} 2$

Beweis $(k \rightarrow k+1)$**:**

- es gelte vor der Iteration $n = t \cdot 2^{k+1} + m$
- Fall 2: t ungerade
  1. $k' := k + 1$
  2. $b[k'] = t \bmod 2 = 1 \rightarrow m' := m + 2^{k+1}$
  3. $t' := (t-1)/2$
- nach der Iteration:

$$\begin{aligned} (t-1)/2 \cdot 2^{k+2} + m + 2^{k+1} &= (t-1) \cdot 2^{k+1} + m + 2^{k+1} \\ &= t \cdot 2^{k+1} + m = n \end{aligned}$$

Invarianten

Beispiel III - Schluss

m die Zahl, die durch $b[0,\dots,k]$ repräsentiert wird

Function convertToBinary($n : \mathbb{N}$) :

 $b :=$ **Array** $< 0,\dots,0 >$ // binary representation of n
 $t := n$
 $k := -1$
 while $t > 0$ **do**
  invariant $n = t \cdot 2^{k+1} + m$
  $k := k + 1$
  $b[k] := t \bmod 2$
  $t := t \operatorname{div} 2$
 assert $\sum_{i=0} b[i] \times 2^i = n$ // Nachbedingung **return** b

- nach Beenden der Schleife gilt: $t = 0$ und $n = t \cdot 2^{k+1} + m$
- $\Rightarrow$ Nachbedingung gilt: $\sum_{i=0} b[i] \times 2^i = m = n$
- $\Rightarrow$ Algorithmus liefert korrektes Ergebnis

Teile-und-Herrsche-Methode

Teile-und-Herrsche-Methode

- Viele Algorithmen sind rekursiv aufgebaut
- Werden oft mehrmals ausgeführt, um “eng verwandte” Teilprobleme zu lösen
- Arbeite nach Methode “Teile und Herrsche”

Paradigma von Teile-und-Herrsche [CLRS04]

- Umfasst drei Schritte auf jeder Rekursionsebene:
  - Teile das Problem in eine Anzahl von Teilproblemen auf.
  - Beherrsche die Teilprobleme durch rekursives Lösen. Wenn die Teilprobleme hinreichend klein sind, dann löse diese auf direktem Weg.
  - Verbinde die Lösungen der Teilprobleme zur Lösung des Ausgangsproblem.