OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

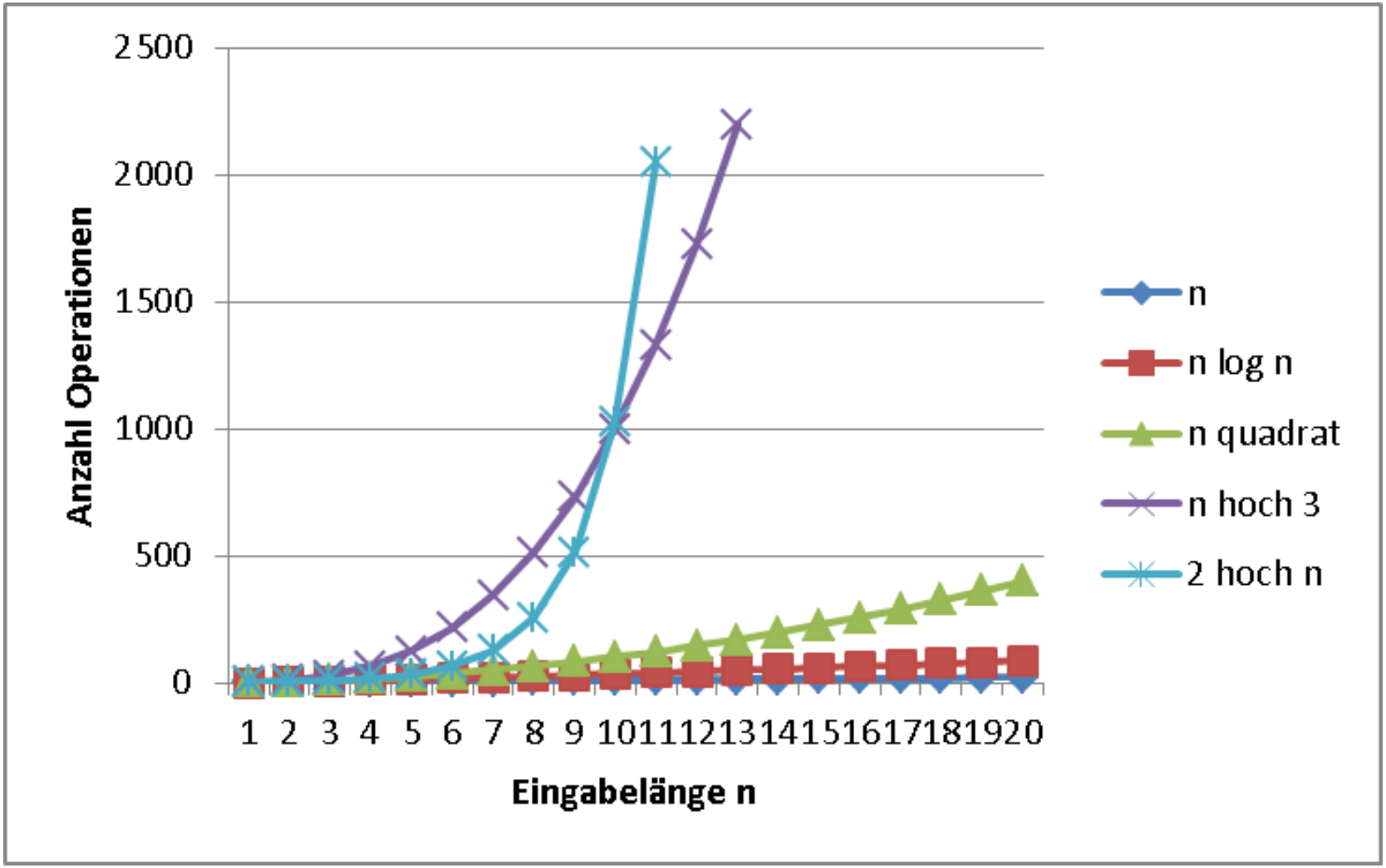
Algorithmen und Datenstrukturen

- Grundlagen (Komplexität) -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 11. Oktober 2018

Wesentliche Wachstumsfunktionen



Beispiel MaxTeilSum

- Eingabe: $a_0, \ldots, a_{n-1} \in \mathbb{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{0 \le i \le j \le n-1} \sum_{k=i}^{j} a_k$$

- Anwendungen
  - Erkennung von grafischen Mustern
  - Analyse von Aktienkursen
    (täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
  - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
  - Ausgabe: 75 (ergibt sich aus $i = 1, j = 5$)

MaxTeilSum1

- Naiver Algorithmus: Durchlaufen aller Varianten

```
int MaxTeilsum1(int a[],int n) {
      int i, j, k, sum, max = int.MinValue;

      for (i=0; i<n; i++) {
            for (j=i; j<n; j++) {
                  sum=0;
                  for (k=i; k<=j; k++) sum += a[k];
                  if (sum > max) max = sum;
            }
      }

      return max;
}
```

- Laufzeit: $T_1{}^{wc}(n) = O(n^3)$ (kubisch)

Laufzeitanalyse MaxTeilSum1

- Zu verarbeitende Eingabedaten: n ganze Zahlen
- Laufzeit in Abhängigkeit der Eingabe sei ${T_1}^{wc}(n)$, dann gilt:

$$ {T_1}^{wc}(n) = 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left(\sum_{k=i}^{j} 1 + 2 \right) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 3) = \sum_{i=0}^{n-1} \sum_{k=0}^{n-1-i} (k + 3) $$

$$ = \sum_{i=0}^{n-1} \sum_{k=0}^{i} k + \sum_{i=0}^{n-1} \sum_{k=0}^{i} 3 = \sum_{i=0}^{n-1} \frac{i(i+1)}{2} + \sum_{i=0}^{n-1} 3(i+1) $$

$$ = \frac{1}{2} \left(\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i \right) + 3 \sum_{i=0}^{n-1} i + n $$

$$ = \frac{1}{2} \left(\frac{(n-1)n(2(n-1)+1)}{6} + \frac{(n-1)n}{2} \right) + \frac{3(n-1)n}{2} = \ldots = \Theta(n^3) $$

MaxTeilSum2

- Verbesserung: Statt immer wieder vollständige Summe zu berechnen: Erweiterung um den aktuellen Summanden

```
int MaxTeilsum2(int a[],int n) {
      int i, j, sum, max = int.MinValue;

      for (i=0; i<n; i++) {
            sum=0;
            for (j=i; j<n; j++) {
                  sum += a[j];
                  if (sum > max) max = sum;
            }
      }

      return max;   }
```

- Laufzeit: $T_2{}^{wc}(n) = O(n^2)$ (quadratisch)

MaxTeilSum3

- Verbesserung: Berechne lokales und globales Maximum. Das globale Maximum ist das Maximum über alle lokalen Maxima. Das lokale Maximum ist an einer Position entweder der Wert oder der Wert plus Werte „von links“.

```
int MaxTeilsum3(int a[],int n) {
      int i, s, max = int.MinValue, aktSum = 0;

      for (i=0; i<n; i++) {
            s = aktSum + a[i];
            if (s > a[i]) aktSum = s;
            else aktSum = a[i];
            if (aktSum > max) max = aktSum;
      }

      return max;   }
```

- Laufzeit: $T_3{}^{wc}(n) = O(n)$ (linear)

Beispiel MaxTeilSum3

Folge	i	aktSum	max
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11		0	-2147483648
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	0	-13	-13
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	1	25	25
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	2	59	59
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	3	71	71
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	4	68	71
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	5	75	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	6	-12	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	7	28	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	8	-49	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	9	11	75

MaxTeilSum3 (Korrektheit I)

- Behauptung:
  - Nach dem i.ten Schleifendurchlauf gilt (Schleifeninvariante):

$$\text{aktSum} = \max_{0 \le l \le i} \sum_{k=l}^{i} a_k\,, \qquad \text{max} = \max_{0 \le l \le m \le i} \sum_{k=l}^{m} a_k$$

- IA: $i = 0$:
  - In der Schleife wird $s = \text{aktSum} + a[i] = 0 + a[0] = a[0]$
  - Da s nicht größer als $\text{a[i]} = a[0]$ wird $\text{aktSum} = \text{a[i]} = \text{a[0]}$
  - 1. Fall: $\text{aktSum} > \text{max}$, dann wird $\text{max} = \text{aktSum}$
  - 2. Fall: aktSum ist kleinster int-Wert,
    dann hatte max den Wert schon bei Initialisierung
- IV: Behauptung gilt für i-1

MaxTeilSum3 (Korrektheit II)

- IS: $i-1 \to i$:
  - Nach IV gilt nach dem $(i-1)$.ten Schleifendurchlauf (Schleifeninvariante):

$$\text{aktSum} = \max_{0 \le l \le i-1} \sum_{k=l}^{i-1} a_k \text{ und } \max = \max_{0 \le l \le m \le i-1} \sum_{k=l}^{m} a_k$$

  - Es wird $s = \text{aktSum} + a[i]$
  - 1. Fall: $s > a[i]$, dann $\text{aktSum} = s$
  - 2. Fall: sonst, dann $\text{aktSum} = a[i]$

$$\Big\} \quad \text{aktSum} = \max_{0 \le l \le i} \sum_{k=l}^{i} a_k$$

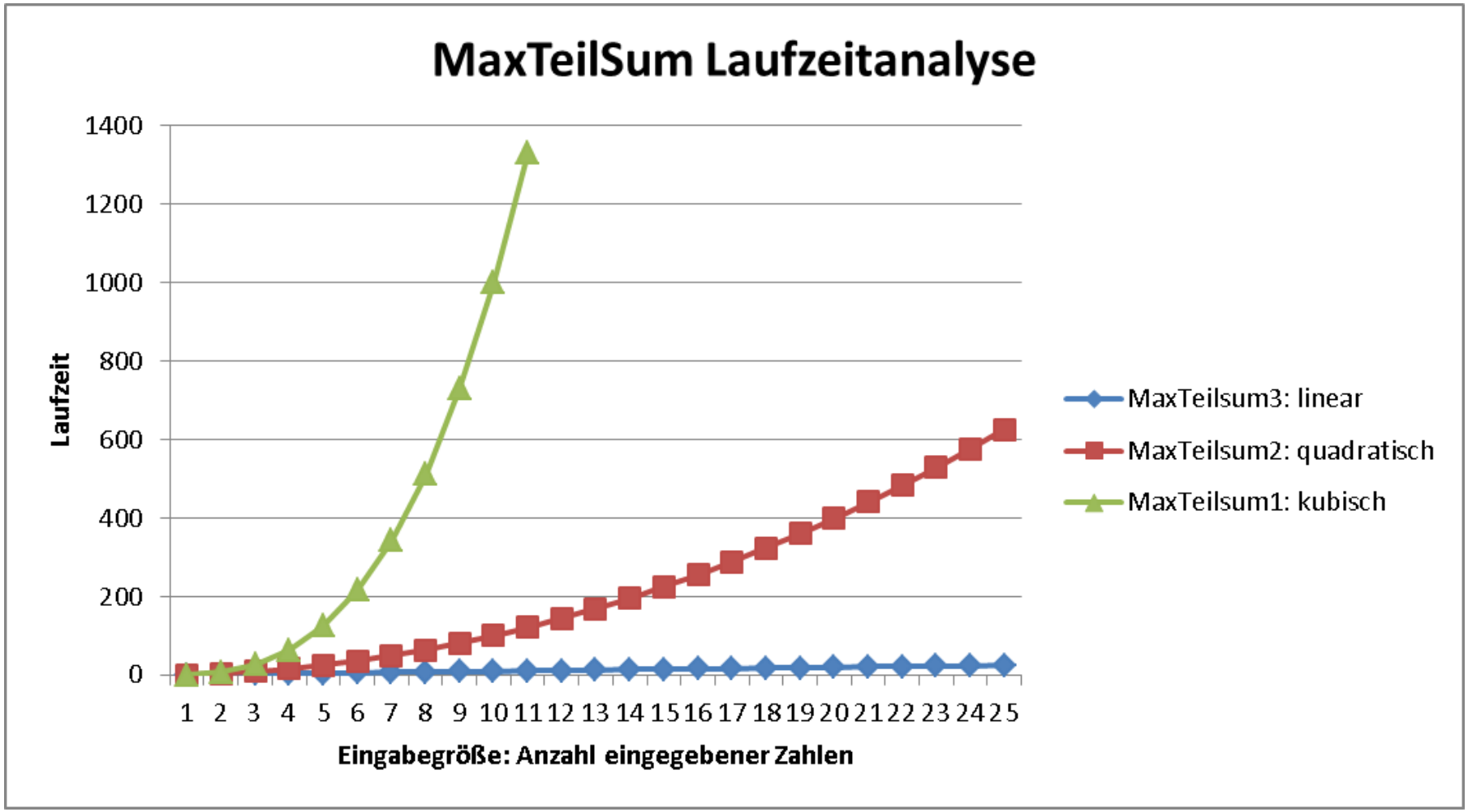
  - Falls $\text{aktSum} > \max$, dann $\max = \text{aktSum}$, d.h.

$$\max = \max \left\{ \max_{0 \le l \le m \le i-1} \sum_{k=l}^{m} a_k \, , \text{aktSum} \right\} = \max_{0 \le l \le m \le i} \sum_{k=l}^{m} a_k$$

  - Nach dem letzten Schleifendurchlauf gilt: $i = n-1$, d.h.

$$\max = \max_{0 \le l \le m \le i} \sum_{k=l}^{m} a_k = \max_{0 \le l \le m \le n-1} \sum_{k=l}^{m} a_k$$

Übersicht
Laufzeitverhalten MaxTeilSum



Komplexität von Algorithmen

- Ein Algorithmus A hat die Komplexität $O(f(n))$, wenn

$$T_A{}^{wc}(n) = O(f(n))$$

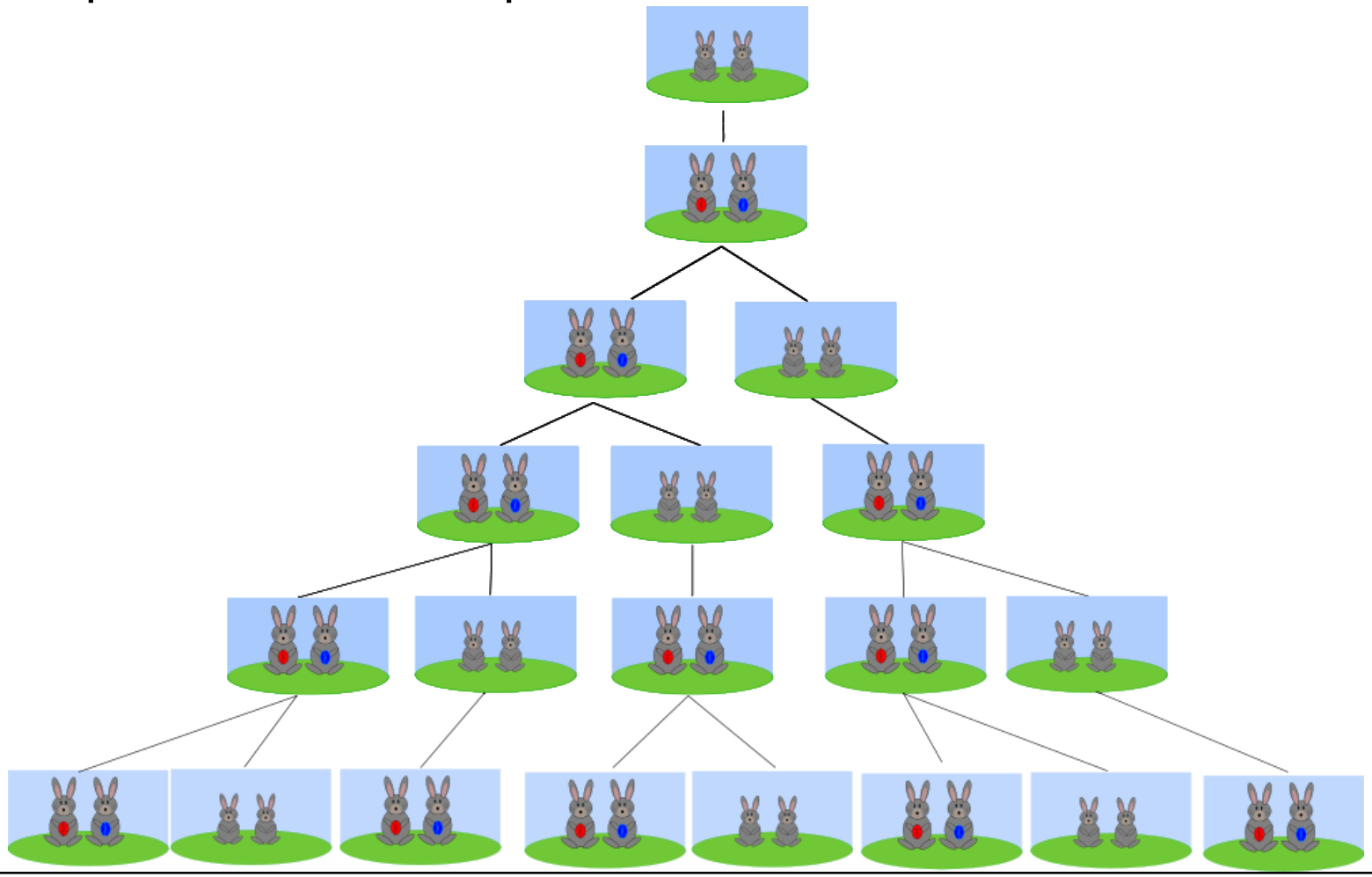
- Komplexität von nun an: Worst Case Laufzeitkomplexität
- Statt $T_A{}^{wc}(n)$ wird auch gerne einfach nur $T(n)$ verwendet
- Vorgehen
  - Bestimme $T(n)$ für den Algorithmus möglichst exakt (Bestimmung genauer Konstanten ist oft schwierig/aufwendig)
  - Schätze $T(n)$ mit Hilfe der O-Notation ab
- Allgemein gilt:
  - Je kleiner die obere Schranke, desto besser
  - Je größer die untere Schranke, desto besser

Übersicht Komplexitätsklassen

Menge	Laufzeit	Beispiele
$O(1)$	Konstant	Abfrage eines Wertes an einer bestimmten Stelle in einem Feld
$O(\log n)$	Logarithmisch	Binäre Suche in einem Feld (Halbierungsprinzip)
$O(n)$	Linear	Suche eines Wertes in einem unsortierten Feld, Fibonacci iterativ
$O(n \log n)$	Super-Linear	Sortieren mit guten Algorithmen (z.B. Mergesort)
$O(n^k)$	Quadratisch, Kubisch, … Allgemein: Polynomiell	Sortieren mit einfachen Algorithmen (z.B. Bubblesort)
$O(2^n)$	Exponentiell	Rekursive Variante zur Berechnung der Fibonacci-Folge, SAT
$O(n!)$	Faktoriell	Problem des Handlungsreisenden (Traveling Salesperson Problem, TSP)

Fibonacci-Zahlen

- Beispiel: Kaninchen-Population



Iterative vs. Rekursive Algorithmen I

- Beispiel: Fibonacci-Zahlen (Kaninchen-Population)
  - $\text{fib}(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{sonst} \end{cases}$ $(\text{fib}(n) = 1,1,2,3,5,8,13, \ldots)$
- Iterative Implementierung

```
int fib(int n) {
        int i=2,result=1,f1=1,f2=1;
        while (i<n)
        {
                i=i+1;
                result=f1+f2;
                f1 = f2;
                f2 = result;
        }
        return result; }
```

Laufzeit: $T(n) = \Theta(n)$

Speicher: $S(n) = \Theta(1)$

Iterative vs. Rekursive Algorithmen II

- Rekursive Implementierung

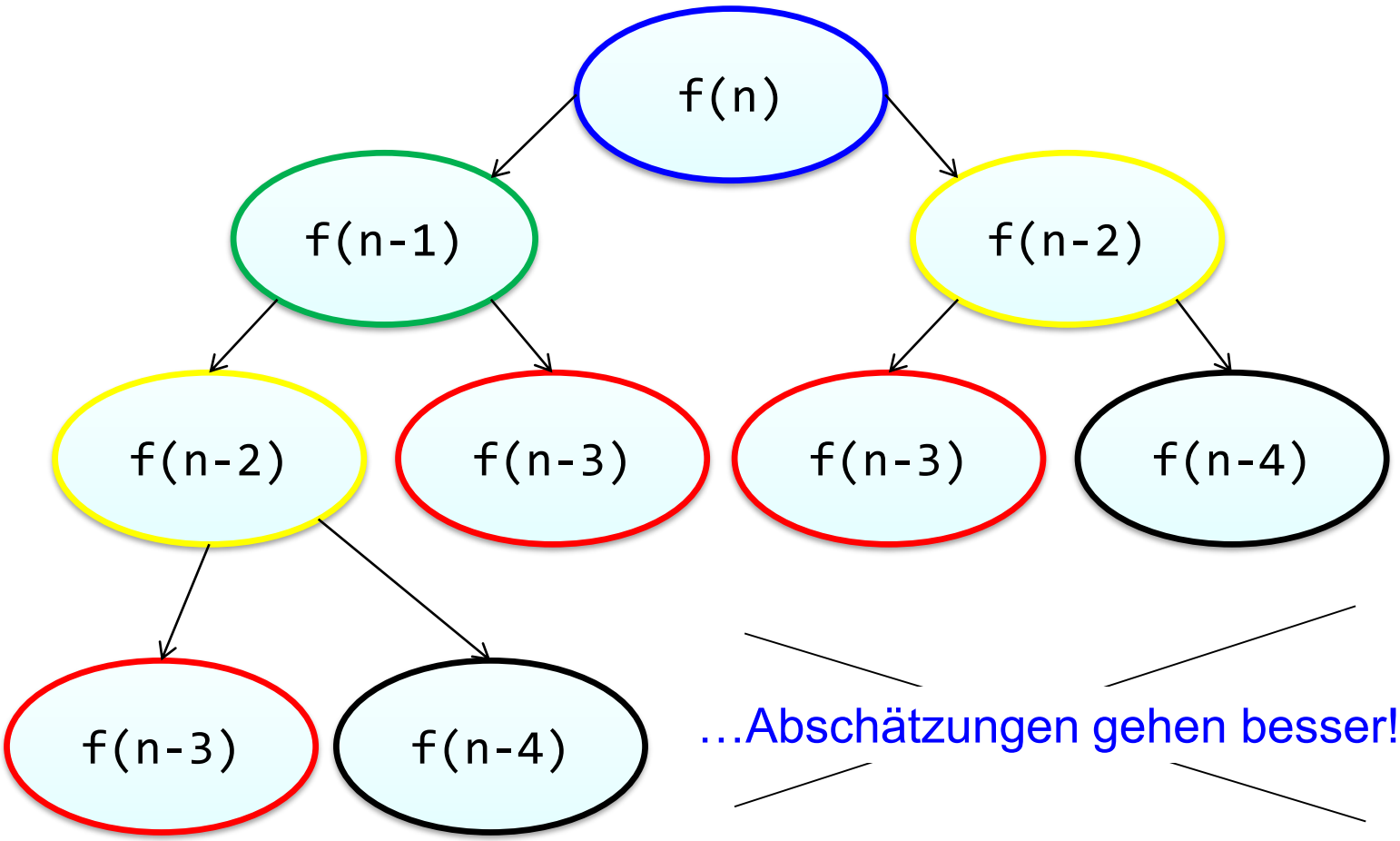
```
int fib_rek(int n) {
        if (n<3) return 1;
        else return fib_rek(n-1) + fib_rek(n-2);
}
```

- Laufzeitanalyse (Substitutionsmethode)

$$T(1) = 1, T(2) = 1, T(n) = T(n-1) + T(n-2)$$

- Behauptung: $T(n) \leq 2^n = O(2^n)$
  - IA: $T(1) = 1 \leq 2^1 = O(2^n), T(2) = 1 \leq 2^2 = O(2^n)$
  - IV: Die Behauptung gilt für n-1 und n-2
  - IS: $T(n) = T(n-1) + T(n-2) \leq^{IV} 2^{n-1} + 2^{n-2} < 2 \cdot 2^{n-1} = O(2^n)$
- Wie kommt man auf die Behauptung?

Rekursionsbaum



≤ 1 Aufruf

≤ 2 Aufrufe

≤ 4 Aufrufe

≤ 8 Aufrufe

$\leq 2^i$ Aufrufe

Maximal 2^n Aufrufe mit konstantem Aufwand, Speicher: $O(2^n)$

Anmerkungen

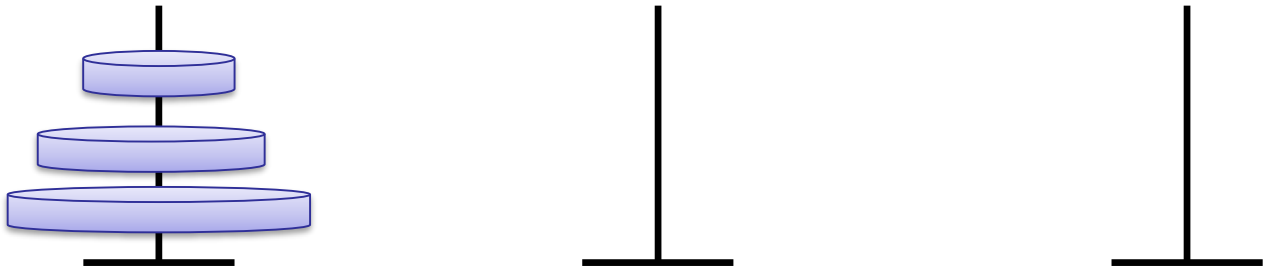
- Komplexität rekursive Implementierung Fibonacci
  - Man kann zeigen (Übungsaufgabe), dass gilt:

$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

- Rekursion ist elegant, sollte aber vermieden werden
- Iterative Varianten sind meist effizienter, aber komplizierter
- Funktionale Sprachen (z.B. Haskell, ML, SML) versuchen Probleme der Rekursion in imperativen Sprachen zu lösen
- Jede rekursive Implementierung kann in eine iterative Implementierung überführt werden
- Aus didaktischen Gründen wird weiter Rekursion verwendet, auch wenn sie nicht unbedingt sinnvoll ist

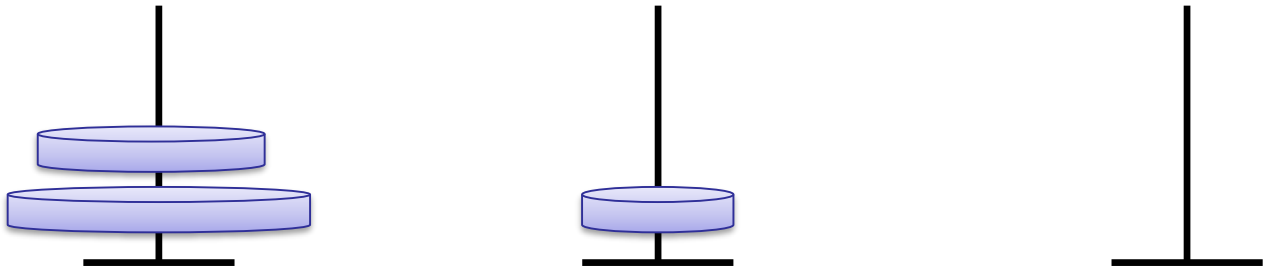
Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einen kleineren Ring abgelegt werden



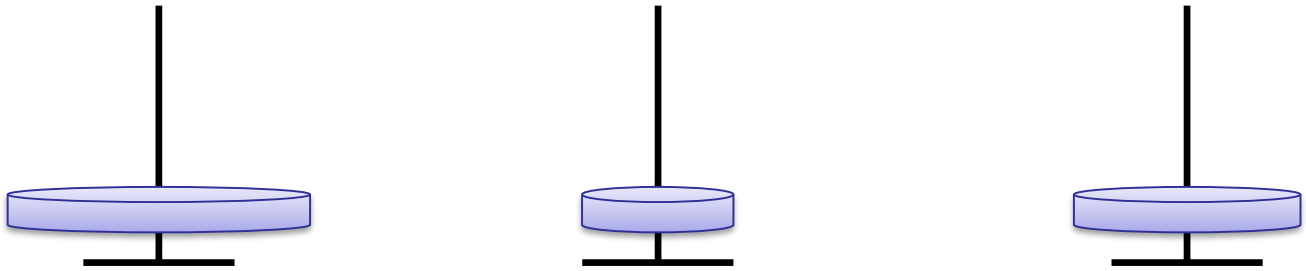
Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



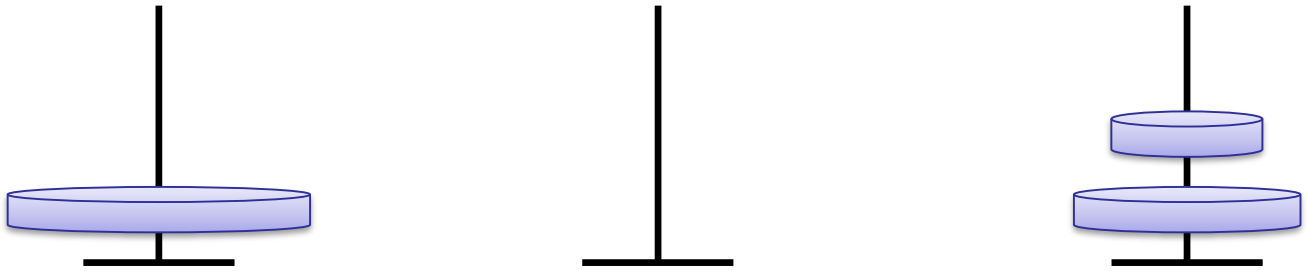
Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



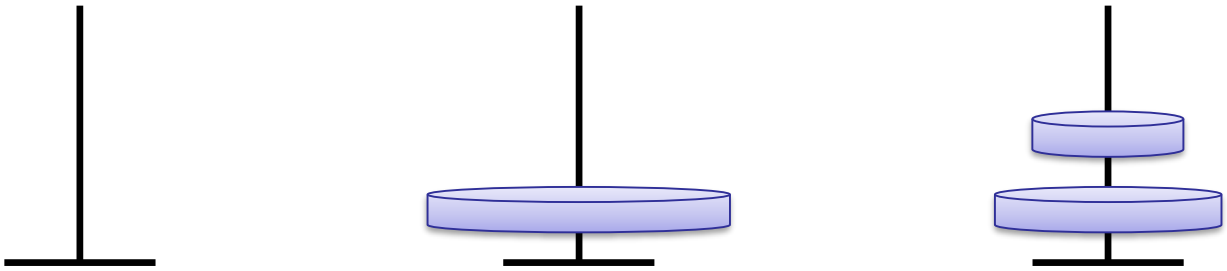
Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



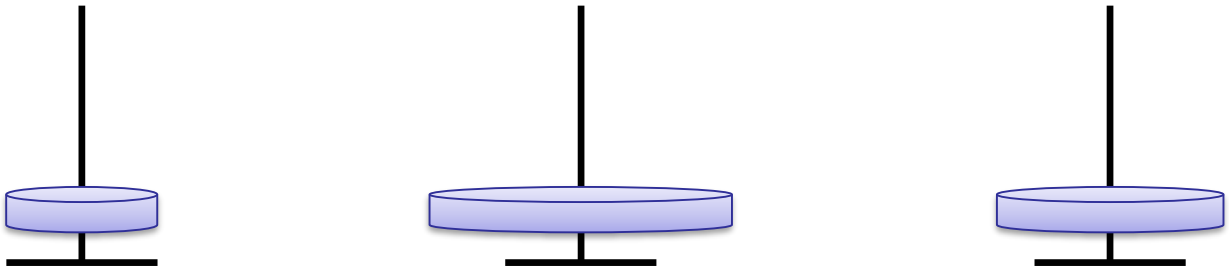
Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



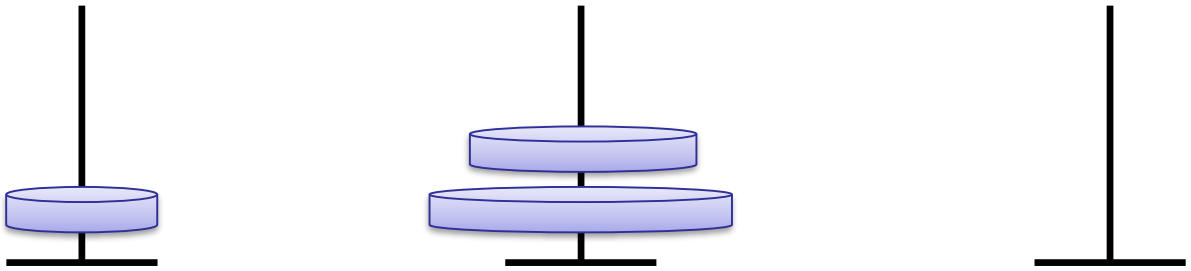
Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



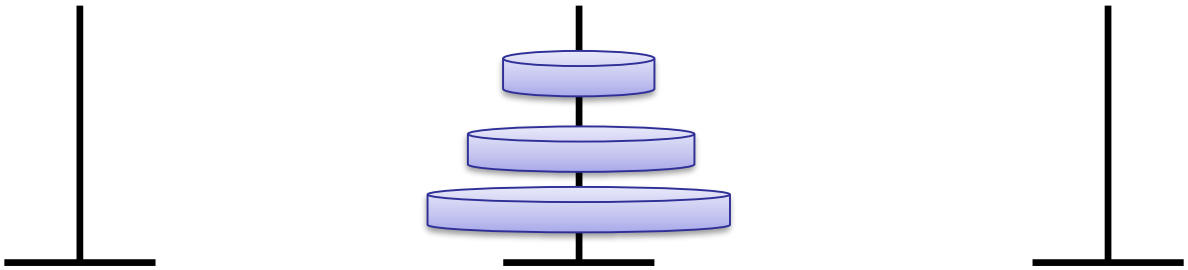
Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



Beispiel: Türme von Hanoi

- Eingabe:
  - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
  - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
  - Es darf immer nur der oberste Ring von einem Stab versetzt werden
  - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



Türme von Hanoi (Rekursiver Algorithmus)

```
void hanoi(int n, char stab1[], char stab2[], char stab3[])
{
    if (n==1)
        // Bewege den obersten Ring von Stab 1 auf Stab 2
    else
    {
        hanoi(n-1,stab1,stab3,stab2);
        // Bewege den obersten Ring von Stab 1 auf Stab 2
        hanoi(n-1,stab3,stab2,stab1);
    }}
```

- Korrektheit (Beh.: n Ringe werden korrekt von Stab 1 auf Stab 2 bewegt)
  - IA: $n = 1$: Der oberste Ring wird von Stab 1 auf Stab 2 bewegt
  - IV: Die Behauptung gilt für $n - 1$
  - IS: $n - 1 \rightarrow n$: Nach IV werden $n - 1$ Ringe von Stab 1 auf Stab 3 bewegt (Stab 2 dient als Hilfe). Der n.te Ring wird von Stab 1 auf Stab 2 bewegt. Zuletzt werden nach IV $n - 1$ Ringe von Stab 3 auf Stab 2 bewegt (Stab 1 dient als Hilfe). D.h. n Ringe wurden von Stab 1 auf Stab 2 bewegt

Türme von Hanoi (Laufzeit)

- Beobachtung Rekursionsgleichungen
  - $T(1) = 1, T(n) = 2T(n-1) + 1$
- Iterationsmethode
  - $T(n) = 2T(n-1) + 1$
    $= 2(2T(n-2) + 1) + 1 = 2^2 T(n-2) + (1+2)$
    $= 2(2(2T(n-3) + 1) + 1) + 1 = 2^3 T(n-3) + (1+2+4)$
    $\dots$
    $= 2^i T(n-i) + \sum_{k=0}^{i-1} 2^k$
  - Rekursionsbasis wird erreicht, wenn: $n - i = 1$, d.h. setze $i = n-1$
  - $T(n) = 2^{n-1} T(n-(n-1)) + \sum_{k=0}^{(n-1)-1} 2^k$
    $= 2^{n-1} + \frac{2^{(n-2)+1}-1}{2-1} = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 = \Theta(2^n)$
- Beweis durch vollständige Induktion (Übung)

Beispiel MaxTeilSum

- Eingabe: $a_0, \ldots, a_{n-1} \in \mathbb{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{0 \le i \le j \le n-1} \sum_{k=i}^{j} a_k$$

- Anwendungen
  - Erkennung von grafischen Mustern
  - Analyse von Aktienkursen
    (täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
  - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
  - Ausgabe: 75 (ergibt sich aus $i = 1, j = 5$)

MaxTeilSum4 (Divide-&-Conquer)

```
int MaxTeilsum4(int a[],int f, int l) { int n = l - f + 1;
   if (n == 1) return a[f];
     else {
        int newn = (n % 2 == 0 ? n / 2 : n / 2 + 1);

        int MaxBorderSum1=a[f+newn-1],i=f+newn-2,currVal=MaxBorderSum1;
        while (i>=f) { currVal+=a[i];
                if (currVal>MaxBorderSum1) MaxBorderSum1=currVal;
                i--; }

        int MaxBorderSum2=a[f+newn],i=f+newn+1,currVal=MaxBorderSum2;
        while (i<=l) { currVal+=a[i];
                if (currVal>MaxBorderSum2) MaxBorderSum2=currVal;
                i++; }

        return max(MaxTeilsum4(a,f,f+newn-1),
                max(MaxTeilsum4(a,f+newn,l),MaxBorderSum1+
                                        MaxBorderSum2)); }}
```

Divide: Teile a in zwei Teile

Triviallösung

Conquer: Berechne die Teillösungen

Merge: Füge die Einzelergebnisse zusammen

- Laufzeit: $T(n) = \Theta(n \log n)$

Beispiel MaxTeilSum4 I

Folge	MBS1	MBS2	Σ	MT1	MT2	Max
-13 25 34 12 -3 | 7 -87 28 -77 11	68	7	75	?	?	?
-13 25 34 | 12 -3 7 -87 28 -77 11	59	12	71	?	?	?
-13 25 | 34 12 -3 7 -87 28 -77 11	25	34	59	?	?	?
-13 | 25 34 12 -3 7 -87 28 -77 11	-13	25	12	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11						-13
-13 25 34 12 -3 7 -87 28 -77 11						25
-13 | 25 34 12 -3 7 -87 28 -77 11	-13	25	12	-13	25	25
-13 25 34 12 -3 7 -87 28 -77 11						34
-13 25 | 34 12 -3 7 -87 28 -77 11	25	34	59	25	34	59
-13 25 34 12 | -3 7 -87 28 -77 11	12	-3	9	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11						12

Beispiel MaxTeilSum4 II

Folge	MBS1	MBS2	Σ	MT1	MT2	Max
-13 25 34 12 -3 7 -87 28 -77 11						-3
-13 25 34 12 | -3 7 -87 28 -77 11	12	-3	9	12	-3	12
-13 25 34 | 12 -3 7 -87 28 -77 11	59	12	71	59	12	71
-13 25 34 12 -3 7 -87 28 | -77 11	28	-66	-38	?	?	?
-13 25 34 12 -3 7 -87 | 28 -77 11	-80	28	-52	?	?	?
-13 25 34 12 -3 7 | -87 28 -77 11	7	-87	-80	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11						7
-13 25 34 12 -3 7 -87 28 -77 11						-87
-13 25 34 12 -3 7 | -87 28 -77 11	7	-87	-80	7	-87	7
-13 25 34 12 -3 7 -87 28 -77 11						28
-13 25 34 12 -3 7 -87 | 28 -77 11	-80	28	-52	7	28	28

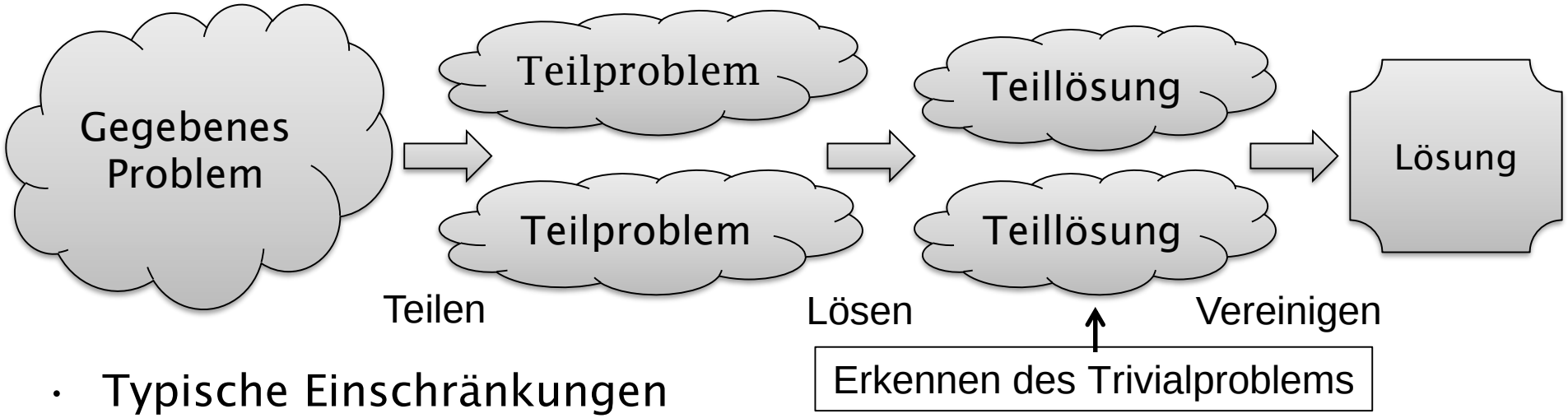
Beispiel MaxTeilSum4 III

Folge	MBS1	MBS2	Σ	MT1	MT2	Max
-13 25 34 12 -3 7 -87 28 | -77 11	-77	11	-66	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11						-77
-13 25 34 12 -3 7 -87 28 -77 11						11
-13 25 34 12 -3 7 -87 28 | -77 11	-77	11	-66	-77	11	11
-13 25 34 12 -3 7 -87 28 | -77 11	28	-66	-38	28	11	28
-13 25 34 12 -3 | 7 -87 28 -77 11	68	7	75	71	28	75

- …und jetzt mit:
  - Eingabe: -5, 13, -32, 7, -3, 17, 23, 12, -35, 19
  - Ausgabe: ?

Entwurfsprinzip: Divide & Conquer

- Schema: Teilen und Herrschen



- Typische Einschränkungen
  - Teilprobleme müssen unabhängig voneinander lösbar sein
  - Gesamtlösung muss aus Teillösungen entstehen können (Vereinigung)
- Teilung bis zum Erreichen des Trivialproblems (oft rekursiv)
- Beispiele
  - Quicksort, Schnelle Fouriertransformation (FFT)

Anmerkungen

- Korrektheitsbeweise häufig mit vollständiger Induktion
  - Identifikation einer Bedingung, die nach allen Schleifendurchläufen gilt (Schleifeninvariante, Analyse: vor, während, nach)
  - Bedingung für das Verlassen der Schleife zusammen mit der Schleifeninvariante liefern das gewünschte Ergebnis
- Komplexitätsabschätzung durch Aufstellen von
  - Komplexitätsgleichungen ($T(n) = \ldots$)
  - Rekursionsgleichungen mit Rekursionsbasis ($T(n) = \ldots, T(a) = b$)
- Lösen von Rekursionsgleichungen durch
  - Substitutionsmethode (Lösung raten, Korrektheit beweisen)
  - Iterationsmethode (sukzessives Einsetzen liefert Abschätzung)
  - Master-Methode (folgt)