Algorithmen und Datenstrukturen

Stefan Roth, SS 2025

02

Sortieren

Folien beruhen auf der Veranstaltung von Prof. Marc Fischlin und Christian Janson aus dem SS 2024

Laufzeitanalysen: $\mathcal{O}$-Notation

Laufzeitanalyse

Wieviel Schritte macht Algorithmus
in Abhängigkeit von der Eingabekomplexität?

meistens: schlechtester Fall über
alle Eingaben gleicher Komplexität

(Worst-Case-)Laufzeit
$T(n)$ = max { Anzahl Schritte für x }

Maximum über alle Eingaben
x der Komplexität n

fasst alle Eingaben
ähnlicher Komplexität zusammen

Beispiel: n Anzahl zu
sortierender Zahlen

(man könnte auch zusätzlich
Größe der Zahlen betrachten;
wird aber meist von Anzahl dominiert)

Laufzeitanalyse Insertion Sort (I)

n Anzahl zu sortierender Elemente

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
      // insert A[i] in pre-sorted sequence A[0..i-1]
2     key=A[i];
3     j=i-1; // search for insertion point backwards
4     WHILE j>=0 AND A[j]>key DO
5         A[j+1]=A[j]; // move elements to right
6         j=j-1;
7     A[j+1]=key;
```

Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile i hat Aufwand ci

Laufzeitanalyse Insertion Sort (II)

n Anzahl zu sortierender Elemente

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
       // insert A[i] in pre-sorted sequence
2      key=A[i];
3      j=i-1; // search for insertion point b
4      WHILE j>=0 AND A[j]>key DO
5          A[j+1]=A[j]; // move elements to r
6          j=j-1;
7      A[j+1]=key;
```

Zeile	Aufwand	Anzahl
1	c1	n
2	c2	$n-1$
3	c3	$n-1$
4	c4	**Z5**$+n-1$
5	c5	$n(n-1)/2$
6	c6	$n(n-1)/2$
7	c7	$n-1$

Zeilen **4**, **5** und **6** im schlimmsten Fall bis `j=-1` also jeweils `i`-mal. Insgesamt:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

(Zeile **4** jeweils einmal mehr bis Abbruch)

Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile i hat Aufwand ci

Laufzeitanalyse Insertion Sort (III)

n Anzahl zu sortierender Elemente

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
       // insert A[i] in pre-sorted sequence
2      key=A[i];
3      j=i-1; // search for insertion point b
4      WHILE j>=0 AND A[j]>key DO
5          A[j+1]=A[j]; // move elements to r
6          j=j-1;
7      A[j+1]=key;
```

Zeile	Aufwand	Anzahl
1	c1	n
2	c2	$n-1$
3	c3	$n-1$
4	c4	**Z5**$+n-1$
5	c5	$n(n-1)/2$
6	c6	$n(n-1)/2$
7	c7	$n-1$

maximale Gesamtlaufzeit Insertion-Sort:

$$T(n) = c1 \cdot n + (c2 + c3 + c4 + c7) \cdot (n-1)$$
$$+(c4 + c5 + c6) \cdot \frac{n(n-1)}{2}$$

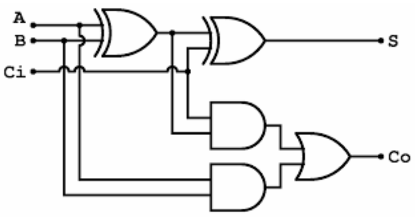
Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile i hat Aufwand ci

Kosten für individuelle Schritte?

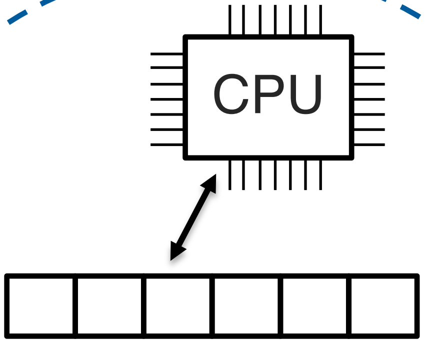
Wie teuer ist z.B. Zuweisung `A[j+1]=A[j]` in Zeile **5**, also was ist $c5$?

Hängt stark von Berechnungsmodell ab
(in dem Pseudocode-Algorithmus umgesetzt wird)…

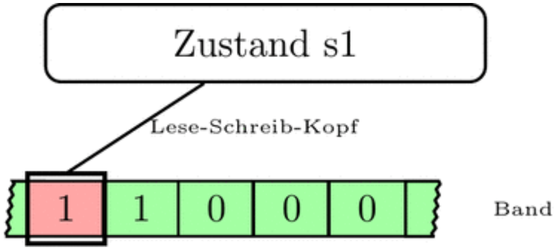


Quelle: ke.tu-darmstadt.de

Schaltkreis



RAM



Quelle: de.wikipedia.org/wiki/Turingmaschine

Turingmaschine

nehmen üblicherweise an, dass elementare Operationen (Zuweisung, Vergleich,…) in einem Schritt möglich → $c5 = 1$

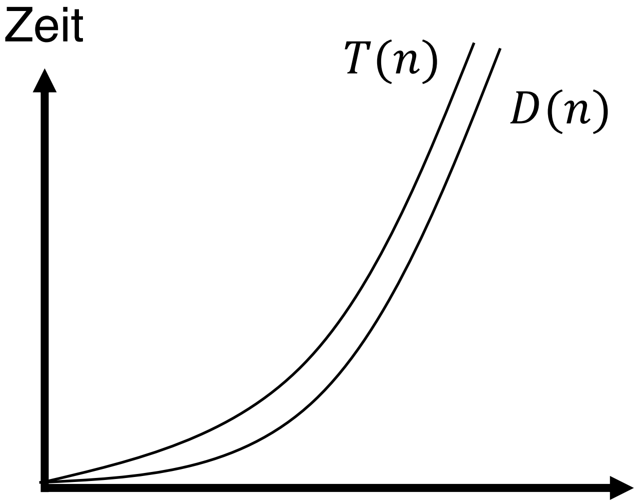
Asymptotische Vereinfachung (I)

Gesamtlaufzeit Insertion-Sort (mit $ci = 1$):

$$T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2}$$

Zum Vergleich (nur dominanter Term) :

$$D(n) = 3 \cdot \frac{n(n-1)}{2}$$



n	$T(n)$	$D(n)$	**relativer Fehler** $(T(n) - D(n))/T(n)$
100	15.346	14.850	3,2321 %
1.000	1.503.496	1.498.500	0,3323 %
10.000	150.034.996	149.985.000	0,0333 %
100.000	15.000.349.996	14.999.850.000	0,0033 %
1.000.000	150.000.034.999.996	1.499.998.500.000	0,0003 %

Asymptotische Vereinfachung (II)

Weiter Vereinfachung (nur abhängiger Term) :

$$A(n) = \frac{n(n-1)}{2}$$

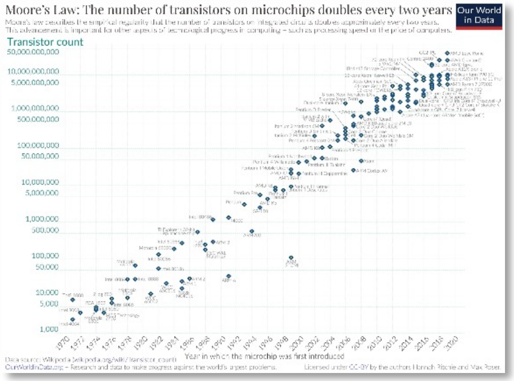
Zum Vergleich (nur dominanter Term) :

$$D(n) = \boxed{3} \cdot \frac{n(n-1)}{2}$$

Konstante hängt stark vom Berechnungsmodell ab

Konstante ändert sich „schnell“ durch Fortschritte in Rechenleistung

Beispiel Moore‘s Law (bis ca. 2000):
Verdoppelung der Transistoren etwa alle 1,5 Jahre



Quelle: `de.wikipedia.org/wiki/Mooresches_Gesetz`

Θ-Notation / Landau-Symbole

Paul Bachmann
Edmund Landau
ca. 1900

Funktionen $f, g\colon \mathbb{N} \longrightarrow \mathbb{R}_{>0}$

Eingabekomplexität Laufzeit

$$\Theta(g) = \{f : \exists \underbrace{c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}}_{\text{Positive Konstanten}}, \forall n \geq n_0, \underbrace{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}_{\substack{f(n) \text{ wird von } c_1 g(n) \text{ und } c_2 g(n) \\ \text{für hinreichend große } n \\ \text{eingeschlossen}}}\}$$

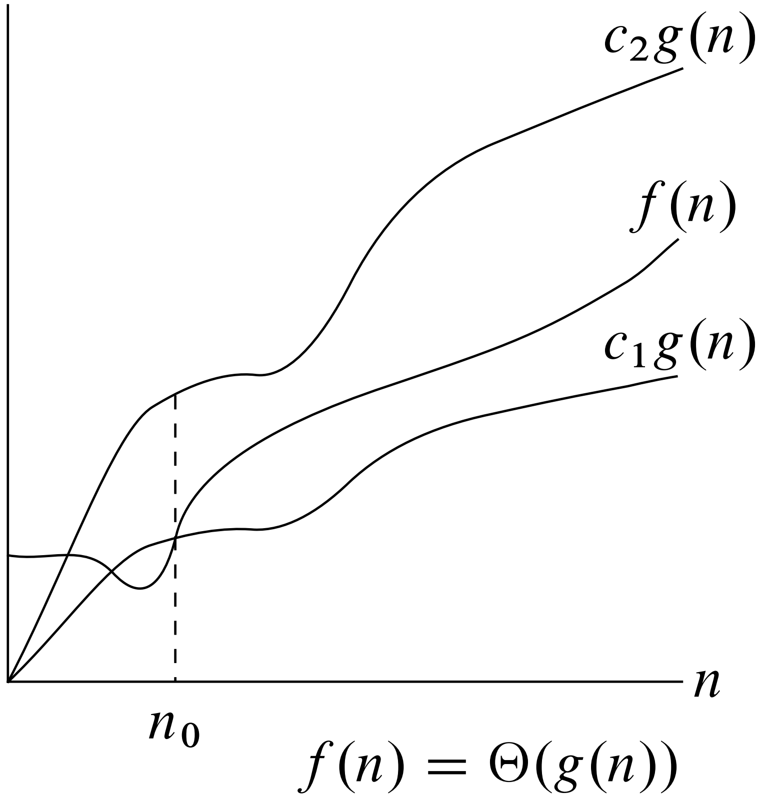
Funktion f

Für alle n größer gleich n_0

Schreibweise: $f \in \Theta(g)$, manchmal auch $f = \Theta(g)$

Veranschaulichung Θ-Notation

Quelle: Corman et al. Introduction to Algorithms



$n \geq n_0$:

$c_1 g(n) \leq f(n)$

$c_2 g(n) \geq f(n)$

$g(n)$ ist eine asymptotisch scharfe Schranke von $f(n)$

Θ-Notation beschränkt eine Funktion asymptotisch von oben und unten

Beispiel: Laufzeit Insertion Sort in Θ-Notation (I)

$$T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2} \quad \in \; \Theta(n^2)$$

Für untere Schranke wähle $c_1 = \frac{3}{2}$ und $n_0 = 2$:

$$\begin{aligned} T(n) &\geq \quad 5 \cdot (n-1) \quad + \frac{3}{2} \cdot n^2 \; - \frac{3}{2} \cdot n \\ &\geq \quad \frac{7}{2} \cdot n - 5 \quad + \frac{3}{2} \cdot n^2 \\ &\geq \quad \frac{3}{2} \cdot n^2 \qquad \text{für } n \geq 2 \end{aligned}$$

$$\Theta(g) = \{f : \exists\, c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Beispiel: Laufzeit Insertion Sort in Θ-Notation (II)

$$T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2} \quad \in \; \Theta(n^2)$$

für $c_1 = \frac{3}{2}$,
$c_2 = 7, n_0 = 2$

Für obere Schranke wähle $c_2 = 7$ und das bereits fixierte $n_0 = 2$:

$$\begin{aligned} T(n) &\le n + 4 \cdot n + 2 \cdot n(n-1) \\ &\le 5 \cdot n + 2 \cdot n^2 \\ &\le 5 \cdot n^2 + 2 \cdot n^2 \\ &\le 7 \cdot n^2 \end{aligned}$$

$$\Theta(g) = \{f : \exists\, c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \ge n_0, 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\}$$

$\mathcal{O}$-Notation

Obere asymptotische Schranke

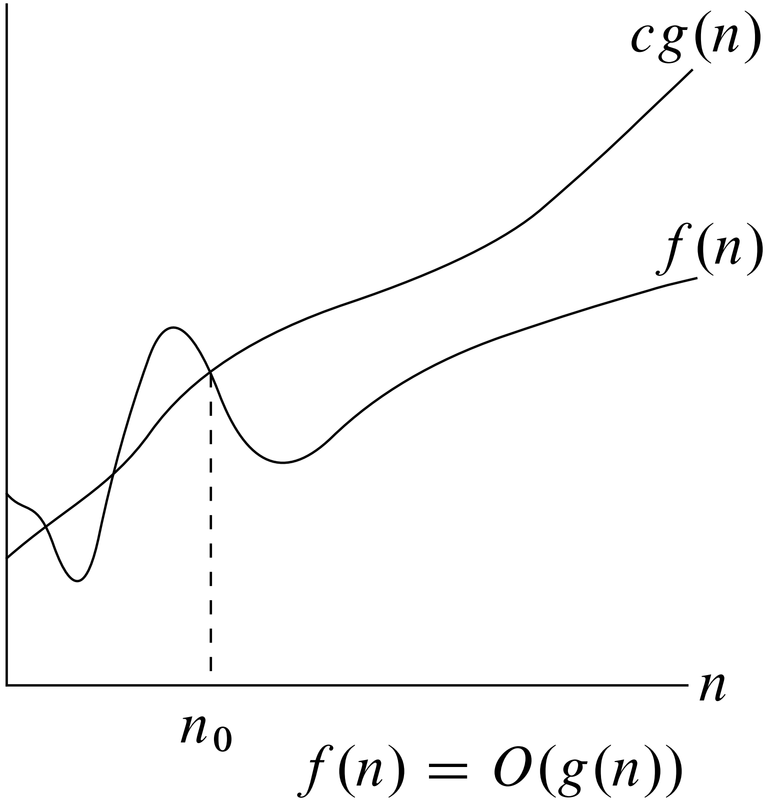
$$\mathcal{O}(g) = \{\underset{\substack{\uparrow\\ \text{Funktion } f}}{f} : \underbrace{\exists\, c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}}_{\text{Positive Konstanten}}, \underset{\substack{\uparrow\\ \text{Für alle } n \text{ größer gleich } n_0}}{\forall n \geq n_0}, \underbrace{0 \leq f(n) \leq cg(n)\}}_{\substack{f(n) \text{ wird von } cg(n)\\ \text{für hinreichend große } n\\ \text{von oben beschränkt}}}$$

Sprechweise: f wächst höchstens so schnell wie g

Schreibweise: $f = \mathcal{O}(g)$ oder auch $f \epsilon\, \mathcal{O}(g)$

Veranschaulichung der $\mathcal{O}$-Notation

Quelle: Corman et al. Introduction to Algorithms



$f(n) = O(g(n))$

$n \geq n_0$:

$0 \leq f(n) \leq cg(n)$

Beachte: $\Theta(g(n)) \subseteq \mathcal{O}(g(n))$ und somit $f(n) = \Theta(g) \implies f(n) = \mathcal{O}(g)$

$\mathcal{O}$-Notation: Rechenregeln

Konstanten: $f(n) = a$ mit $a \in \mathbb{R}_{>0}$ konstant. Dann $f(n) = \mathcal{O}(1)$

Skalare Multiplikation: $f = \mathcal{O}(g)$ und $a \in \mathbb{R}_{>0}$. Dann $a \cdot f = \mathcal{O}(g)$

Addition: $f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2)$. Dann $f_1 + f_2 = \mathcal{O}(\underbrace{\max\{g_1, g_2\}}_{\text{punktweise}})$

Multiplikation: $f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2)$. Dann $f_1 \cdot f_2 = \mathcal{O}(g_1 \cdot g_2)$

Ω-Notation

Untere asymptotische Schranke

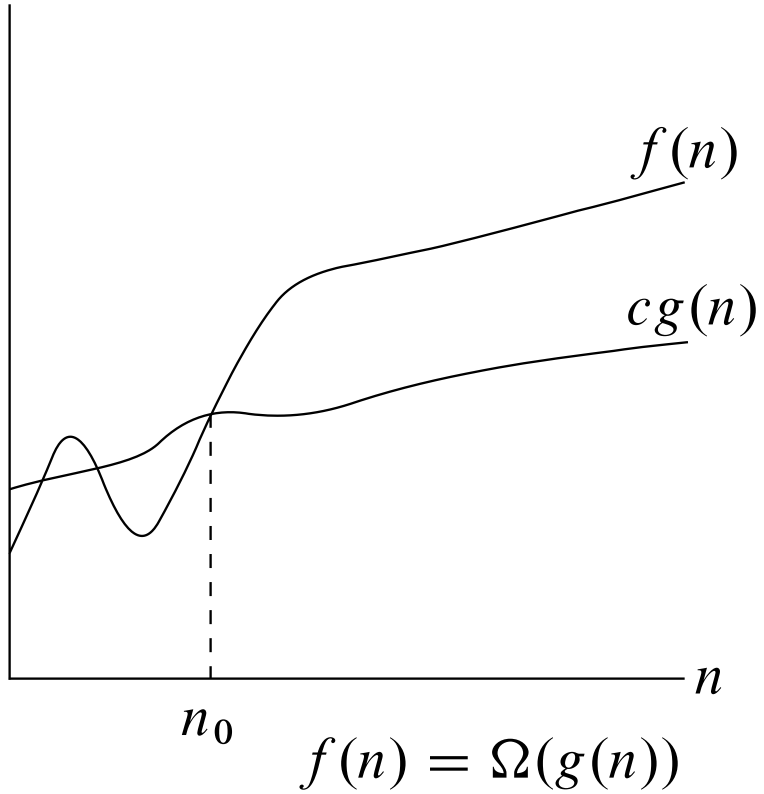
$$\Omega(g) = \{\underset{\substack{\uparrow\\ \text{Funktion } f}}{f} : \underbrace{\exists\, c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}}_{\text{Positive Konstanten}}, \forall n \underset{\substack{\uparrow\\ \text{Für alle } n \text{ größer gleich } n_0}}{\geq} n_0, \underbrace{0 \leq cg(n) \leq f(n)}_{\substack{f(n) \text{ wird von } cg(n)\\ \text{für hinreichend große } n\\ \text{von unten beschränkt}}}\}$$

Sprechweise: f wächst mindestens so schnell wie g

Schreibweise: $f = \Omega(g)$ oder auch $f \in \Omega(g)$

Veranschaulichung der Ω-Notation

Quelle: Corman et al. Introduction to Algorithms



$f(n) = \Omega(g(n))$

$n \geq n_0$:

$0 \leq cg(n) \leq f(n)$

Beachte: $\Theta(g(n)) \subseteq \Omega(g(n))$ und somit $f(n) = \Theta(g) \implies f(n) = \Omega(g)$

Überlegen Sie sich die Rechenregeln für Ω analog zu denen für die $\mathcal{O}$-Notation.

Überlegen Sie sich, dass gilt:

$\mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(n^4) \subseteq \mathcal{O}(n^5) \subseteq \dots$

und

$\Omega(n) \supseteq \Omega(n^2) \supseteq \Omega(n^3) \supseteq \Omega(n^4) \supseteq \Omega(n^5) \supseteq \dots$

und dass die Inklusionen jeweils strikt sind

Zusammenhang $\mathcal{O}, \Omega$ und Θ

Für beliebige $f(n), g(n)\colon \mathbb{N} \longrightarrow \mathbb{R}_{>0}$ gilt:

$$f(n) = \Theta(g(n))$$

genau dann, wenn

$f(n) = \mathcal{O}(g(n))$ und $f(n) = \Omega(g(n))$

Beachte: $\Omega(g), \mathcal{O}(g)$ sind nur untere bzw. obere Schranken:

Beispiel: $f(n) = \Theta(n^3)$, also auch:

$f(n) = \mathcal{O}(n^5)$, da $\Theta(n^3) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(n^5)$

$f(n) = \Omega(n)$, da $\Theta(n^3) \subseteq \Omega(n^3) \subseteq \Omega(n)$

Anwendung $\mathcal{O}$-Notation (I)

$f = \mathcal{O}(g)$ übliche Schreibweise, sollte aber gelesen werden als $f \in \mathcal{O}(g)$

Allgemein besser immer als Mengen auffassen
und von links nach rechts lesen (mit ∈, ⊆):

$$5 \cdot n^2 + n^4 \underset{\in}{=} \mathcal{O}(n^2) + n^4 \underset{\subseteq}{=} \mathcal{O}(n^4) \underset{\subseteq}{=} \mathcal{O}(n^5)$$

als Menge

$$\{f(n)\} + n^4 = \{\, f(n) + n^4 \,\}$$

nicht: $\mathcal{O}(n^4) = \mathcal{O}(n^5)$, und damit auch $\mathcal{O}(n^5) = \mathcal{O}(n^4)$

wird mit Mengenschreibweise klarer: $A \subseteq B$ bedeutet allgemein nicht auch $B \subseteq A$

Anwendung $\mathcal{O}$-Notation (II)

Ungleichungen mit $\leq$ sollten nur mit $\mathcal{O}$ verwendet werden,
Ungleichungen mit $\geq$ sollten nur mit Ω verwendet werden.

$$5 \cdot n^2 + n^4 \leq 6 \cdot n^4 = \mathcal{O}(n^4)$$

es gibt c, n_0 und Funktion $f(n)$ mit $6 \cdot n^4 \leq c \cdot f(n)$

obere Schranke vs. untere Schranke

nicht: $\boldsymbol{5 \cdot n^2 + n^4 \leq 6 \cdot n^4 = \Omega(n^4)}$

$\mathcal{O}, \Omega$ und Θ bei Insertion Sort (I)

```
insertionSort(A)
FOR i=1 TO A.length-1 DO
    // insert A[i] in pre-sorted sequence A[0..i-1]
    key=A[i];
    j=i-1; // search for insertion point backwards
    WHILE j>=0 AND A[j]>key DO
        A[j+1]=A[j]; // move elements to right
        j=j-1;
    A[j+1]=key;
```

Algorithmus macht maximal $T(n)$ viele Schritte und $T(n) = \Theta(n^2)$

Also Laufzeit Insertion Sort $= \Theta(n^2)$?

korrekte Anwendung: Laufzeit $\leq T(n) = \mathcal{O}(n^2)$

$\mathcal{O}, \Omega$ und Θ bei Insertion Sort (II)

```
insertionSort(A)
FOR i=1 TO A.length-1 DO
    // insert A[i] in pre-sorted sequence A[0..i-1]
    key=A[i];
    j=i-1; // search for insertion point backwards
    WHILE j>=0 AND A[j]>key DO
        A[j+1]=A[j]; // move elements to right
        j=j-1;
    A[j+1]=key;
```

zur Erinnerung: (Worst-Case-)Laufzeit $T(n)$ = max { Anzahl Schritte für x }

Für untere Schranke muss man „nur“
eine schlechte bzw. die schlechteste Eingabe x finden

Dann gilt $T(n) \geq$ Anzahl Schritte für schlechtes x

$\mathcal{O}, \Omega$ und Θ bei Insertion Sort (III)

Insertion Sort hat quadratische Laufzeit $\Theta(n^2)$

```
insertionSort(A)
FOR i=1 TO A.length-1 DO
    // insert A[i] in pre-sorted sequence A[0..i-1]
    key=A[i];
    j=i-1; // search for insertion point backwards
    WHILE j>=0 AND A[j]>key DO
        A[j+1]=A[j]; // move elements to right
        j=j-1;
    A[j+1]=key;
```

„Schlechte“ Eingabe:

A	n	$n-1$	$n-2$		$\dots$	$\dots$		2	1

Jede `WHILE`-Schleifenausführung für $i = 1, \dots, n-1$ macht jeweils i Iterationen

Insgesamt macht Algorithmus für **A** also $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$ Schritte

„Gute“ Eingaben für Insertion Sort?

```
insertionSort(A)
FOR i=1 TO A.length-1 DO
    // insert A[i] in pre-sorted sequence A[0..i-1]
    key=A[i];
    j=i-1; // search for insertion point backwards
    WHILE j>=0 AND A[j]>key DO
        A[j+1]=A[j]; // move elements to right
        j=j-1;
    A[j+1]=key;
```

„gute“ Eingaben bereits vorsortiert, extremes Beispiel:

A	1	2	3		…	…		$n-2$	$n-1$

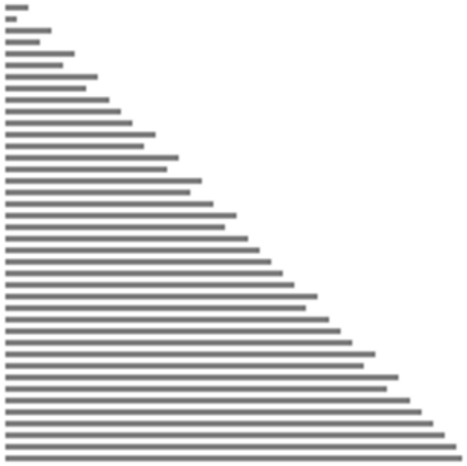
`WHILE`-Schleife wird für dieses **`A`** nie ausgeführt

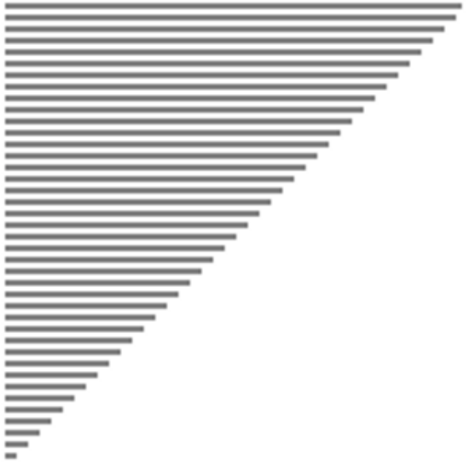
Insgesamt macht Algorithmus für dieses **`A`** also $\Theta(n)$ Schritte

Laufzeit Insertion Sort (I)

„guter“ Fall
(fast vorsortiertes Array)

„schlechter“ Fall
(invertiertes, vorsortiertes Array)



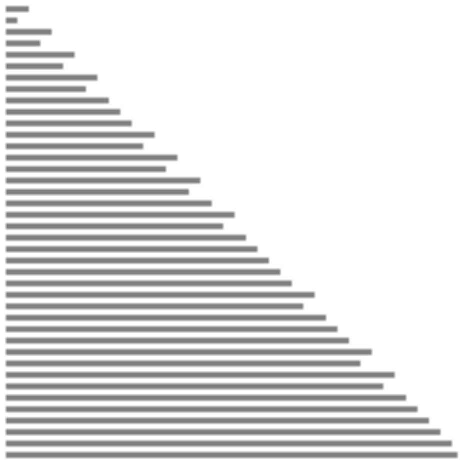


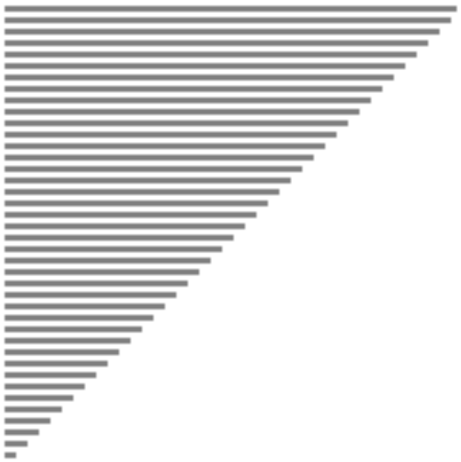
Quelle: `https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/insertion-sort`

Laufzeit Insertion Sort (II)

„guter“ Fall
(fast vorsortiertes Array)

„schlechter“ Fall
(invertiertes, vorsortiertes Array)





Quelle: `https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/insertion-sort`

Wort-Case-Laufzeiten:
auch wenn für manche Eingaben schneller, gilt:

Insertion Sort hat quadratische Laufzeit $\Theta(n^2)$

Komplexitätsklassen

n ist die Länge der Eingabe (z.B. Arraylänge, Länge des Strings)

Klasse	Bezeichnung	Beispiel
$\Theta(1)$	Konstant	Einzeloperation
$\Theta(\log n)$	Logarithmisch	Binäre Suche
$\Theta(n)$	Linear	Sequentielle Suche
$\Theta(n \log n)$	Quasilinear	Sortieren eines Arrays
$\Theta(n^2)$	Quadratisch	Matrixaddition
$\Theta(n^3)$	Kubisch	(naive) Matrixmultiplikation*
$\Theta(n^k)$	Polynomiell	
$\Theta(\mathrm{k}^n)$	Exponentiell	Travelling-Salesperson†
$\Theta(n!)$	Faktoriell	Permutationen

* Strassen-Algorithmus $\mathcal{O}(n^{2.8074})$

† $\Theta(n^2 2^n)$ wenn der Algorithmus geschickt implementiert ist

o-Notation und ω-Notation

nicht asymptotisch scharfe Schranken

$$o(g) = \{f : \underbrace{\forall\, c \in \mathbb{R}_{>0},}_{} \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

Gilt für **alle** Konstanten $c > 0$,
in $\mathcal{O}$-Notation für eine Konstante $c > 0$

Beispiel: $2n = o(n^2)$ und $2n^2 \neq o(n^2)$

$$\omega(g) = \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

Beispiel: $^{n^2}/_2 = \omega(n)$ und $^{n^2}/_2 \neq \omega(n^2)$

Was macht der folgende Sortier-Algorithmus Bubble-Sort?

```
bubbleSort(A)
FOR i=A.length-1 DOWNTO 0 DO
   FOR j=0 TO i-1 DO
      IF A[j]>A[j+1] THEN SWAP(A[j],A[j+1]);
         //temp=A[j+1]; A[j+1]=A[j]; A[j]=temp;
```

Welche Laufzeit hat der Algorithmus?

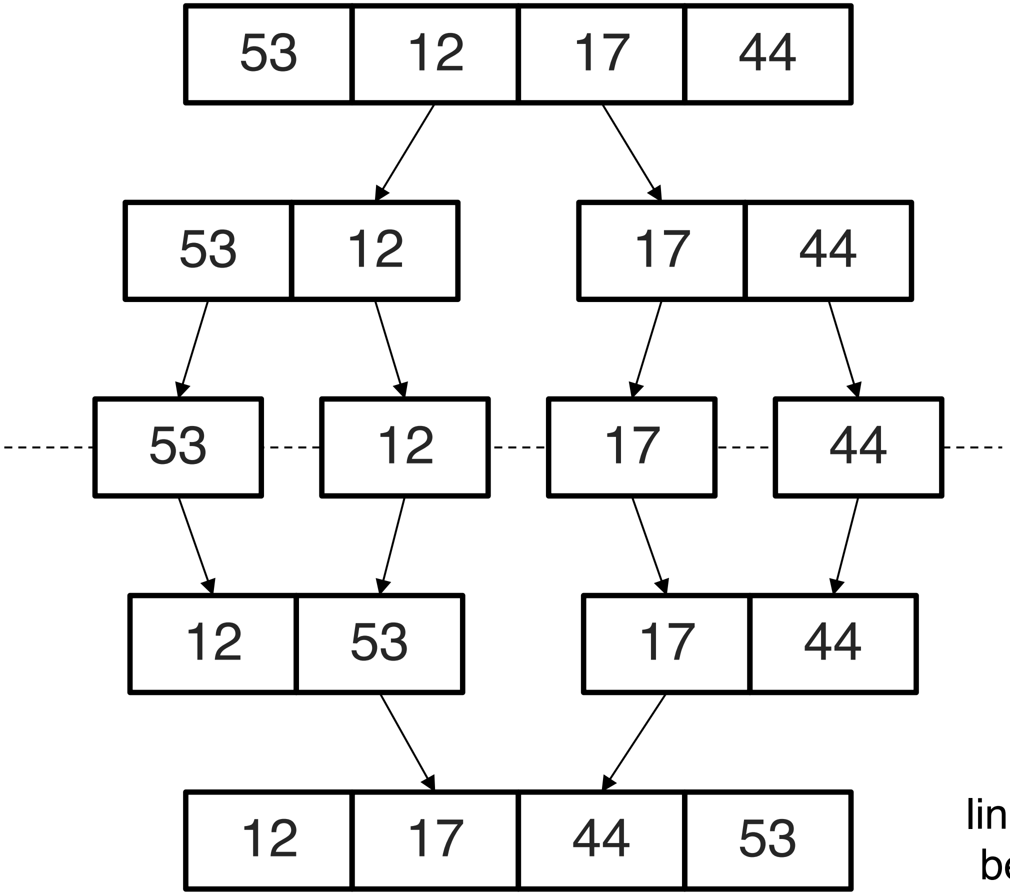
Wie verhält er sich im Vergleich zu Insertion Sort?

Merge Sort

Idee Divide & Conquer (& Combine)

Entwurfsmethoden
→ Abschnitt 7

Teile Liste in Hälften, sortiere (rekursiv) Hälften, sortiere wieder zusammen



(Teil-)Sortierung können im Array selbst erfolgen

`A[0…3]`

`A[0…1], A[2…3]`

`A[0],A[1],A[2],A[3]`

`A[0…1], A[2…3]`

linke & rechte Grenze beschreiben Teilliste

`A[0…3]`

Algorithmus: Merge Sort

Wir sortieren im Array **`A`** zwischen Position **`left`** (links) und **`right`** (rechts)

```
mergeSort(A,left,right) //initial left=0,right=A.length-1

IF left<right THEN  //more than one element
  mid=floor((left+right)/2);  // middle (rounded down)
  mergeSort(A,left,mid);      // sort left part
  mergeSort(A,mid+1,right);   // sort right part
  merge(A,left,mid,right);    // merge into one
```

genauer: letzter Index des linken Teils

$$mid = \left\lceil \frac{right - left + 1}{2} \right\rceil + \frac{2left}{2} - 1 = \left\lceil \frac{right + left - 1}{2} \right\rceil = \left\lfloor \frac{right + left}{2} \right\rfloor$$

Anzahl Elemente /2 (gerundet)

Offset (beginnend mit 0)

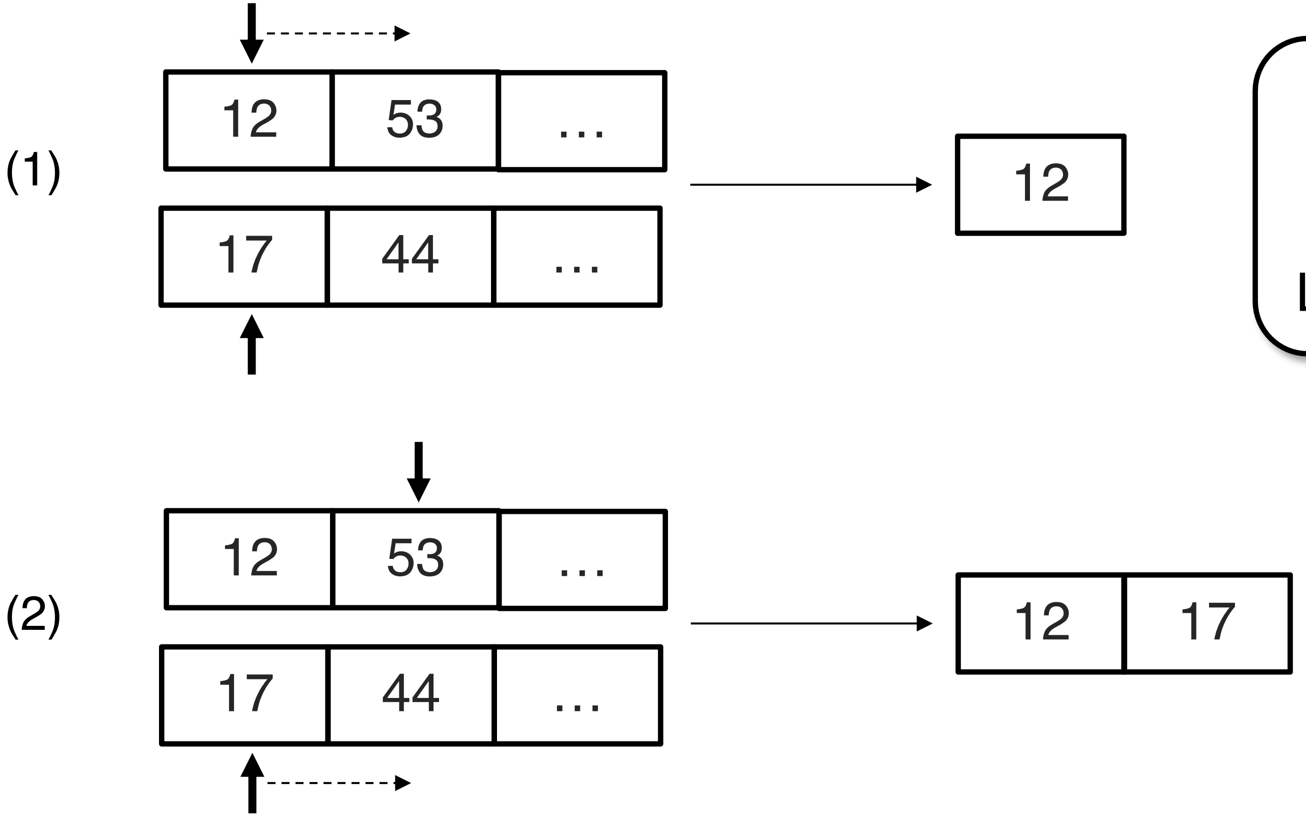
Beispiele:

`left=3, right=4, mid=3`

`left=3, right=5, mid=4`

Merge (für sortierte Teillisten)

Idee: nimm nächstes kleinstes Element aus linker oder rechter Teilliste und gehe in dieser Liste eine Position nach rechts



(1)

12	53	…
17	44	…

→ 12

(2)

12	53	…
17	44	…

→ 12 17

Jede Liste wird genau einmal durchlaufen ⇒ Laufzeit $\Theta(n_L + n_R)$

…

Algorithmus: Merge (für sortierte Teillisten)

rechte Liste noch aktiv und
[linke Liste bereits abgearbeitet oder
nächstes Element rechts]

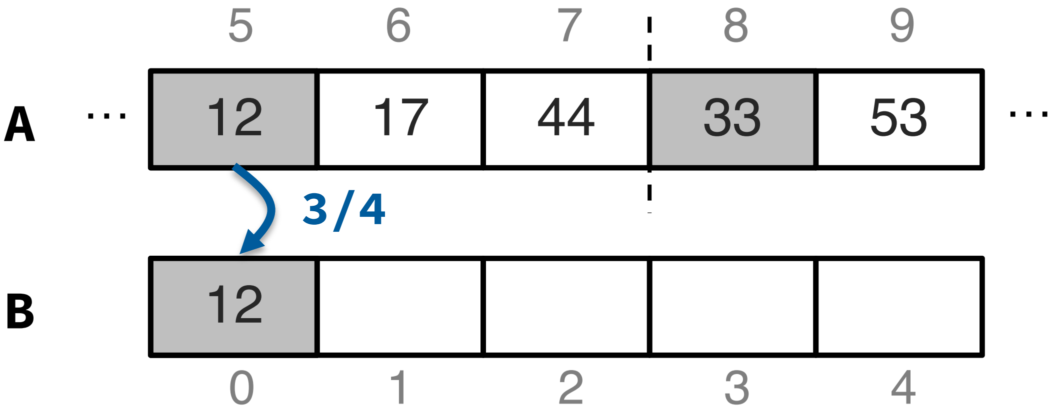
rechte Liste bereits abgearbeitet oder
[linke Liste noch aktiv und nächstes Element links]

```
merge(A,left,mid,right) // requires left<=mid<=right
    //temporary array B, right-left+1 elements

p=left; q=mid+1;             // position left, right
FOR i=0 TO right-left DO     // merge all elements
   IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
      B[i]=A[p];
      p=p+1;
   ELSE  //next element at q
      B[i]=A[q];
      q=q+1;
FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Beispiel: Merge (I)

`left=5, mid=7, right=9`

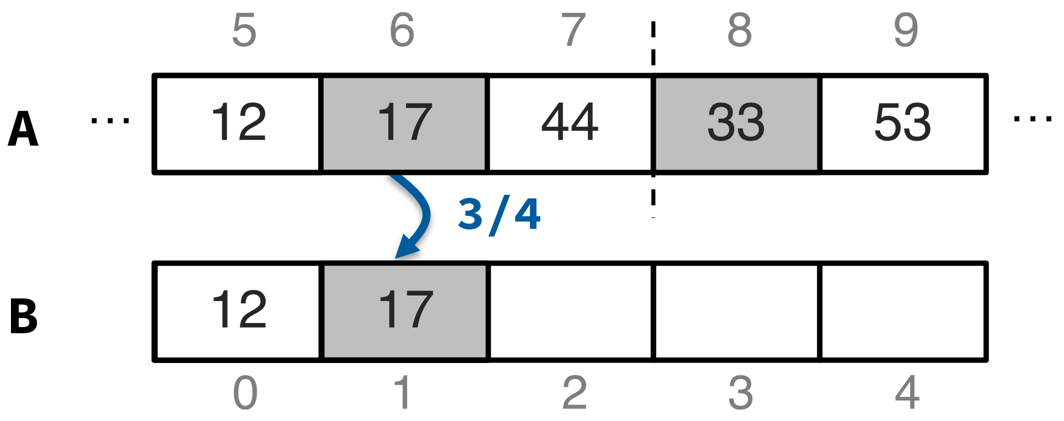


`p=5, q=8, i=0`

```
1 p=left; q=mid+1;               // position left, right
2 FOR i=0 TO right-left DO   // merge all elements
3    IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4       B[i]=A[p];
5       p=p+1;
6    ELSE  //next element at q
7       B[i]=A[q];
8       q=q+1;
9 FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Beispiel: Merge (II)

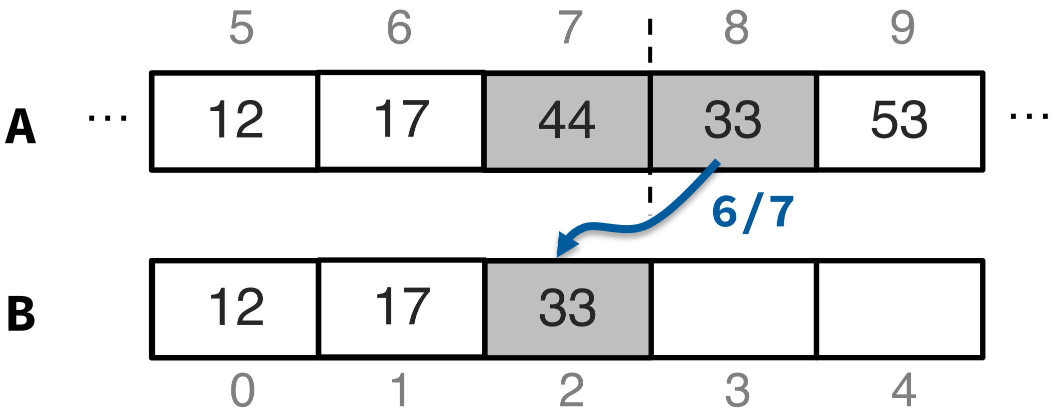
`left=5, mid=7, right=9`



`p=6, q=8, i=1`

```
1 p=left; q=mid+1;              // position left, right
2 FOR i=0 TO right-left DO      // merge all elements
3    IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4       B[i]=A[p];
5       p=p+1;
6    ELSE  //next element at q
7       B[i]=A[q];
8       q=q+1;
9 FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

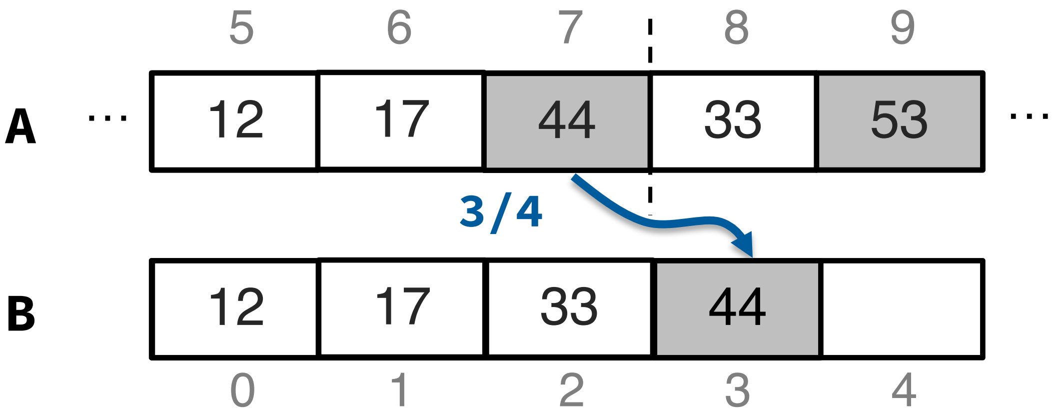
Beispiel: Merge (III) `left=5, mid=7, right=9`



`p=7, q=8, i=2`

```
1 p=left; q=mid+1;          // position left, right
2 FOR i=0 TO right-left DO  // merge all elements
3    IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4       B[i]=A[p];
5       p=p+1;
6    ELSE  //next element at q
7       B[i]=A[q];
8       q=q+1;
9 FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

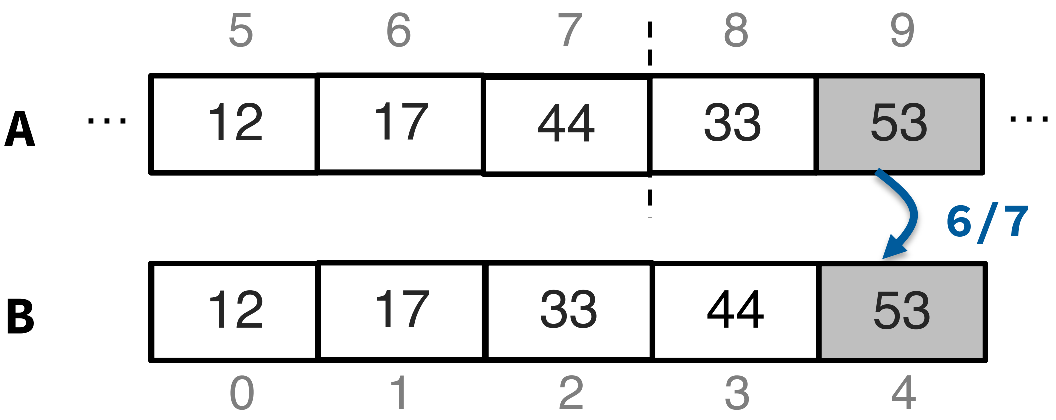
Beispiel: Merge (IV) `left=5, mid=7, right=9`



`p=7, q=9, i=3`

```
1 p=left; q=mid+1;             // position left, right
2 FOR i=0 TO right-left DO     // merge all elements
3    IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4       B[i]=A[p];
5       p=p+1;
6    ELSE  //next element at q
7       B[i]=A[q];
8       q=q+1;
9 FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

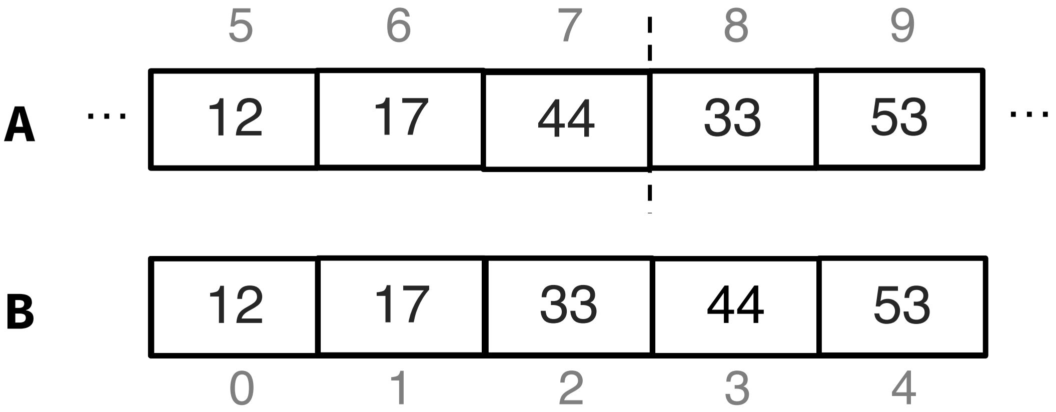
Beispiel: Merge (IV) `left=5, mid=7, right=9`



`p=8, q=9, i=4`

```
1  p=left; q=mid+1;              // position left, right
2  FOR i=0 TO right-left DO      // merge all elements
3     IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4        B[i]=A[p];
5        p=p+1;
6     ELSE  //next element at q
7        B[i]=A[q];
8        q=q+1;
9  FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Beispiel: Merge (IV) `left=5, mid=7, right=9`



		5	6	7	8	9	
A	…	12	17	44	33	53	…

	0	1	2	3	4
B	12	17	33	44	53

`p=8, q=10, i=5`

Ende erste **FOR**-Schleife
und zurückkopieren

```
p=left; q=mid+1;             // position left, right
FOR i=0 TO right-left DO     // merge all elements
   IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
      B[i]=A[p];
      p=p+1;
   ELSE  //next element at q
      B[i]=A[q];
      q=q+1;
FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Korrektheit Merge (I)

(Terminierung klar)

Beweis der Schleifeninvariante per Induktion über **`i`**

Mit entsprechender Interpretation für „leere“ Arrays
und **`B[-1]=`**$-\infty$ sowie **`A[p]=`**$+\infty$ für **`p>mid`** und **`A[q]=`**$+\infty$ für **`q>hi`**

Schleifeninvariante (für Zeile **`2`**): Bei jedem Eintritt (bzw. nach Ende) gilt
`i=p-left+q-(mid+1)`, **`p<=mid+1, q<=right+1,`**
`B[0…i-1]` ist sortiert und besteht aus **`A[left…p-1], A[mid+1…q-1].`**
Ferner : **`B[i-1]<=A[p],A[q]`**

```
2  FOR i=0 TO right-left DO  // merge all elements
3     IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4        B[i]=A[p];
5        p=p+1;
6     ELSE  //next element at q
7        B[i]=A[q];
8        q=q+1;
9  FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Korrektheit Merge (II)

Basisfall **i=0** (**p=left, q=mid+1**)

Gilt mit richtiger Interpretation, da alle Arrays „leer“ und **B[-1]=**–∞

Ferner i=0=p-left+q-(mid+1)
und p<=mid+1, q<=right+1, da left<=mid<=right

Schleifeninvariante (für Zeile **2**): Bei jedem Eintritt (bzw. nach Ende) gilt
i=p-left+q-(mid+1), **p<=mid+1, q<=right+1,**
B[0…i-1] ist sortiert und besteht aus **A[left…p-1], A[mid+1…q-1].**
Ferner : **B[i-1]<=A[p],A[q]**

```
2  FOR i=0 TO right-left DO  // merge all elements
3     IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4        B[i]=A[p];
5        p=p+1;
6     ELSE  //next element at q
7        B[i]=A[q];
8        q=q+1;
9  FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Korrektheit Merge (III)

Induktionsschritt **i−1→i**

Invariante gilt nach Voraussetzung vor (**i−1**)-ter Iteration

Iteration setzt **B[i−1]** auf kleineren bzw. einzigen Wert **A[p],A[q]**

Nach Voraus. **B[i−2]<= A[p],A[q],** so dass **B[i−2]<=B[i−1]**,
also ist **B[0…i−1]** nach Iteration auch sortiert

…

i=p−left+q−(mid+1), **p<=mid+1, q<=right+1,**
B[0…i−1] ist sortiert und besteht aus **A[left…p−1], A[mid+1…q−1].**
Ferner : **B[i−1]<=A[p],A[q]**

```
FOR i=0 TO right-left DO  // merge all elements
   IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
      B[i]=A[p];
      p=p+1;
   ELSE  //next element at q
      B[i]=A[q];
      q=q+1;
FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Korrektheit Merge (IV)

Induktionsschritt **`i-1→i`**

Invariante gilt nach Voraussetzung vor (`i-1`)-ter Iteration

Iteration setzt `B[i-1]` auf kleineren bzw. einzigen Wert `A[p],A[q]`

Da Teil-Arrays vorsortiert, gilt nach **4** bzw. **7** (vor Erhöhen von **p** bzw. **q**):

`B[i-1]=min{A[p],A[q]} <= A[p],A[p+1],A[q],A[q+1]`

…

`i=p-left+q-(mid+1)`, **`p<=mid+1, q<=right+1,`**

`B[0…i-1]` ist sortiert und besteht aus **`A[left…p-1], A[mid+1…q-1].`**

Ferner : **`B[i-1]<=A[p],A[q]`**

```
2  FOR i=0 TO right-left DO  // merge all elements
3     IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4        B[i]=A[p];
5        p=p+1;
6     ELSE  //next element at q
7        B[i]=A[q];
8        q=q+1;
9  FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Korrektheit Merge (V)

Induktionsschritt **`i-1`→`i`**

Invariante gilt nach Voraussetzung vor (**`i-1`**)-ter Iteration

Zähler des kopierten Werts wird erhöht,
also **`B[0…i-1]`** wegen Voraussetzung aus angegebener Menge

…

`i=p-left+q-(mid+1)`, **`p<=mid+1, q<=right+1,`**
`B[0…i-1]` ist sortiert und besteht aus **`A[left…p-1], A[mid+1…q-1].`**
Ferner : **`B[i-1]<=A[p],A[q]`**

```
FOR i=0 TO right-left DO  // merge all elements
   IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
      B[i]=A[p];
      p=p+1;
   ELSE  //next element at q
      B[i]=A[q];
      q=q+1;
FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Korrektheit Merge (VI)

Induktionsschritt `i−1→i`

Invariante gilt nach Voraussetzung vor (`i−1`)-ter Iteration

Zähler `i−1` wird um eins erhöht, und entweder `p` oder `q` auch um eins

Wenn `p>=mid+1` bzw. `q>=right+1` wird die Teilliste nicht mehr gewählt, also Zählerwerte nicht mehr erhöht

`i=p−left+q−(mid+1)`, `p<=mid+1, q<=right+1,`
`B[0…i−1]` ist sortiert und besteht aus `A[left…p−1], A[mid+1…q−1].`
Ferner : `B[i−1]<=A[p],A[q]`

```
FOR i=0 TO right−left DO  // merge all elements
   IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
      B[i]=A[p];
      p=p+1;
   ELSE  //next element at q
      B[i]=A[q];
      q=q+1;
FOR i=0 TO right−left DO A[i+left]=B[i]; //copy back
```

Korrektheit Merge (VII)

Nach Ende der **`FOR`**-Schleife (**`i=right-left+1`**) folgt aus
`i=p-left+q-(mid+1)` und **`p=<mid+1, q=<right+1`**,
dass **`q=right+1`** und **`p=mid+1`**

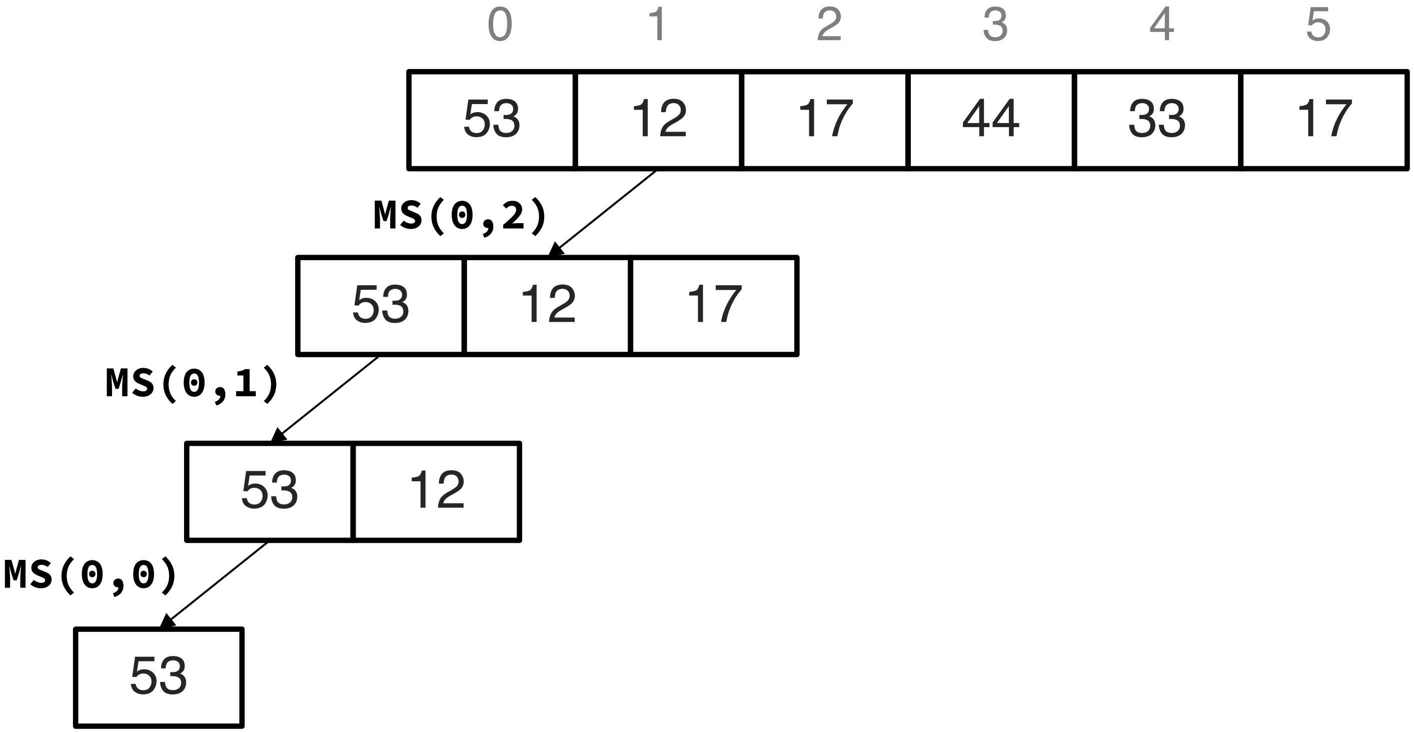
Also besteht **`B[0…right-left]`** aus **`A[left…mid],A[mid+1…right]`**
und ist sortiert

`i=p-left+q-(mid+1)`, **`p<=mid+1, q<=right+1,`**
`B[0…i-1]` ist sortiert und besteht aus **`A[left…p-1], A[mid+1…q-1].`**
Ferner : **`B[i-1]<=A[p],A[q]`**

```
FOR i=0 TO right-left DO  // merge all elements
   IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
      B[i]=A[p];
      p=p+1;
   ELSE  //next element at q
      B[i]=A[q];
      q=q+1;
FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

Beispiel: Merge Sort (I)

```
IF left<right THEN  //more than one element
  mid=floor((left+right)/2); // middle (rounded down)
  mergeSort(A,left,mid);     // sort left part
  mergeSort(A,mid+1,right);  // sort right part
  merge(A,left,mid,right);   // merge into one
```



0	1	2	3	4	5
53	12	17	44	33	17

`MS(0,2)`

53	12	17

`MS(0,1)`

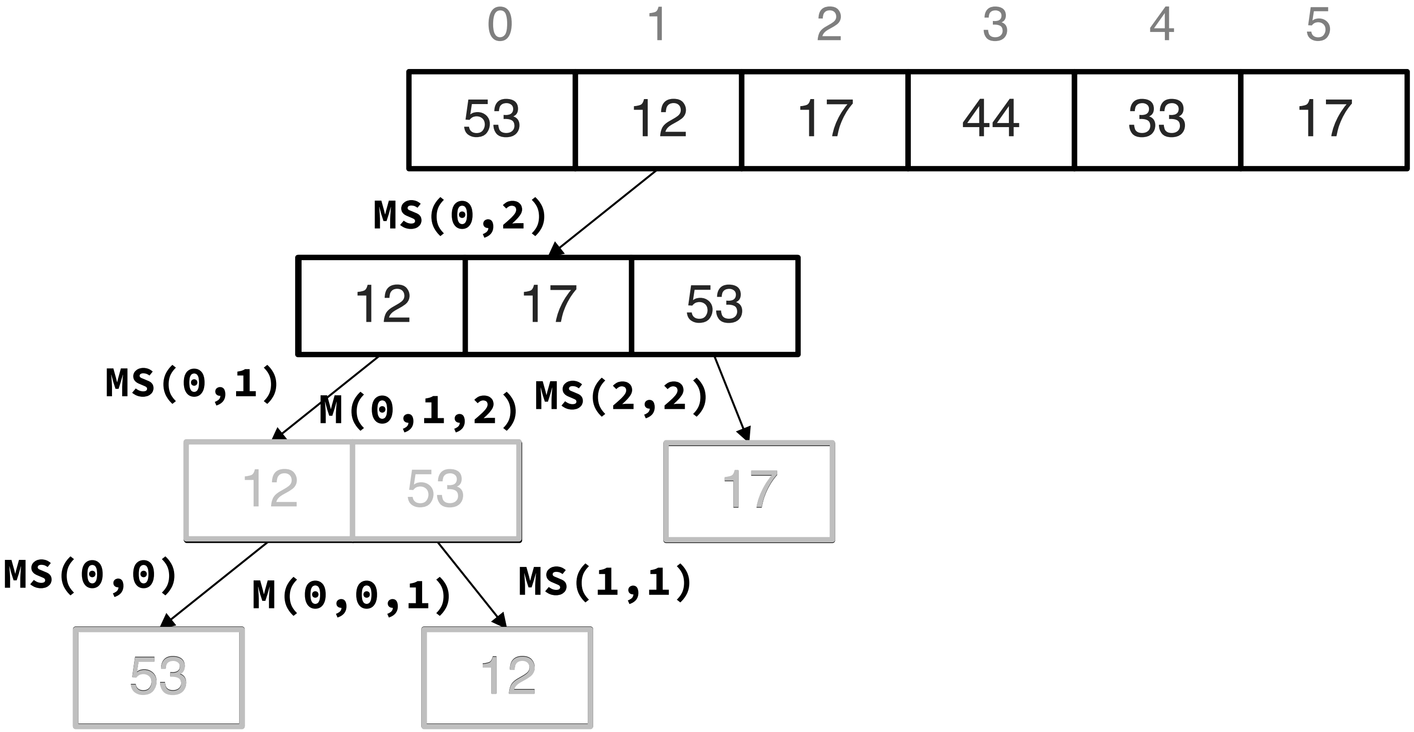
53	12

`MS(0,0)`

53

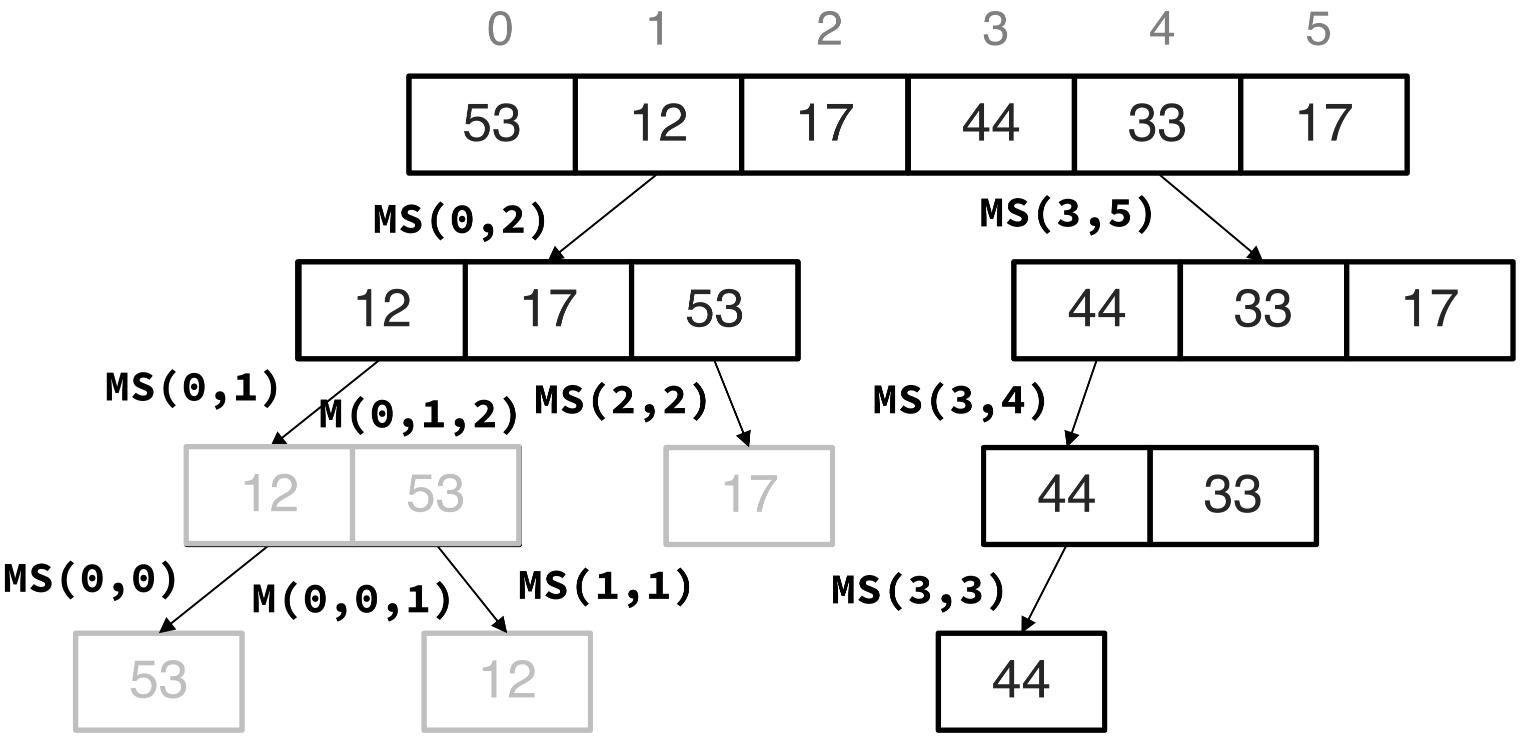
Beispiel: Merge Sort (II)

```
IF left<right THEN  //more than one element
  mid=floor((left+right)/2); // middle (rounded down)
  mergeSort(A,left,mid);     // sort left part
  mergeSort(A,mid+1,right);  // sort right part
  merge(A,left,mid,right);   // merge into one
```



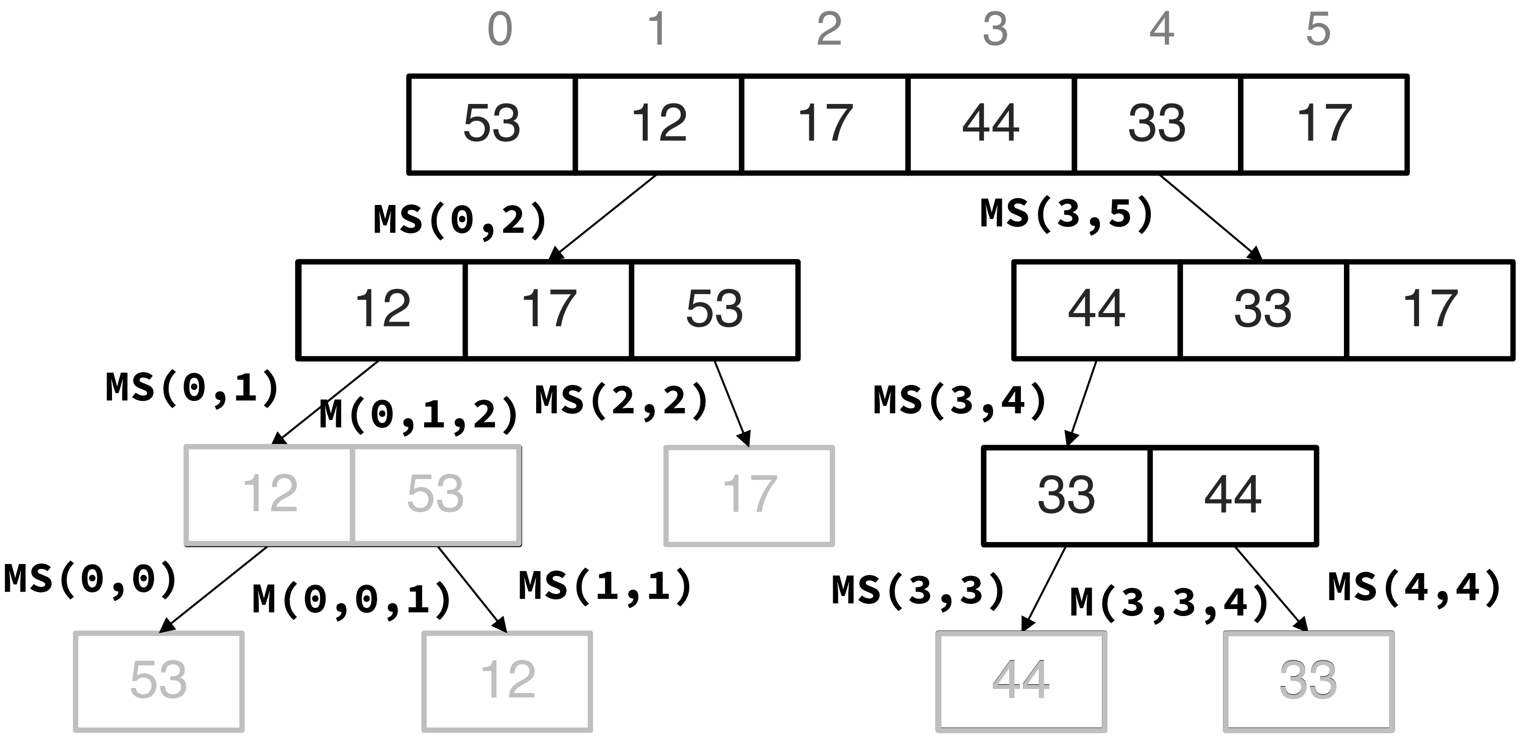
Beispiel: Merge Sort (III)

```
IF left<right THEN  //more than one element
  mid=floor((left+right)/2); // middle (rounded down)
  mergeSort(A,left,mid);     // sort left part
  mergeSort(A,mid+1,right);  // sort right part
  merge(A,left,mid,right);   // merge into one
```



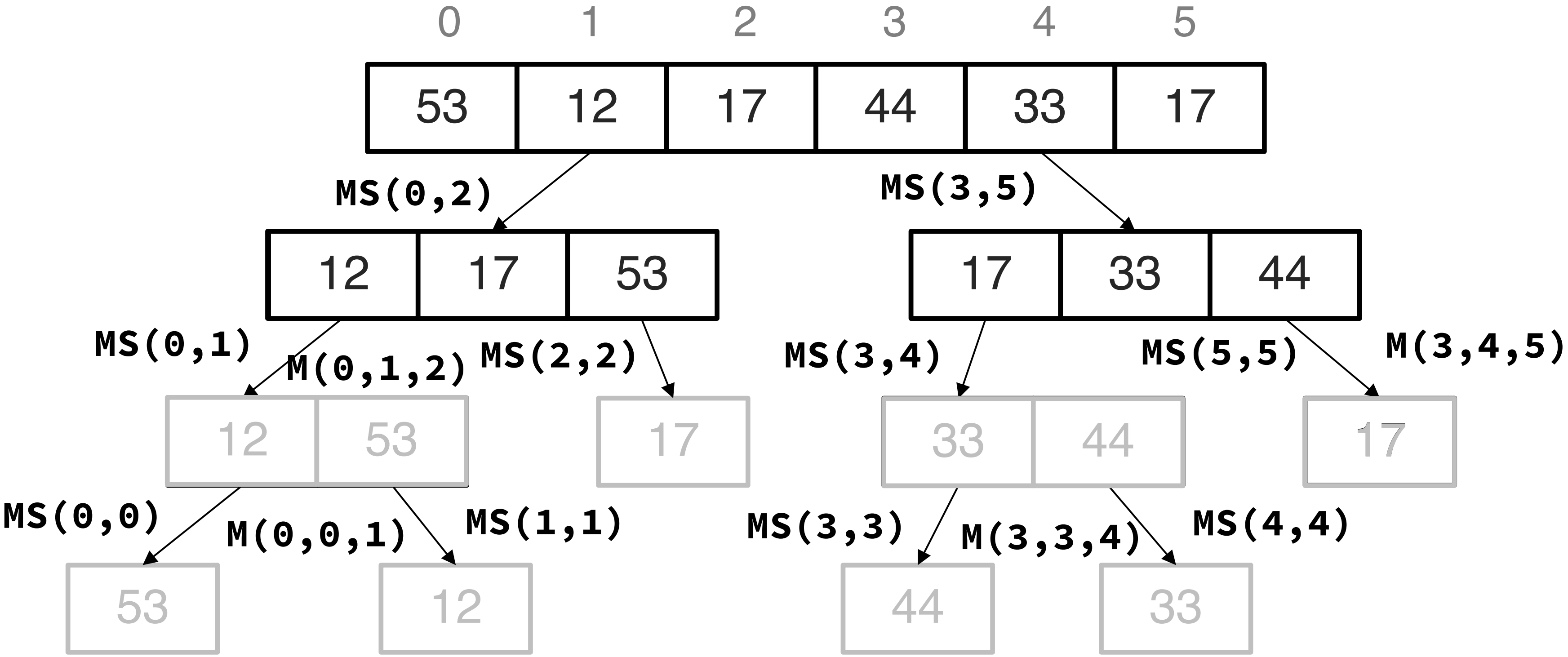
Beispiel: Merge Sort (IV)

```
IF left<right THEN  //more than one element
  mid=floor((left+right)/2); // middle (rounded down)
  mergeSort(A,left,mid);     // sort left part
  mergeSort(A,mid+1,right);  // sort right part
  merge(A,left,mid,right);   // merge into one
```



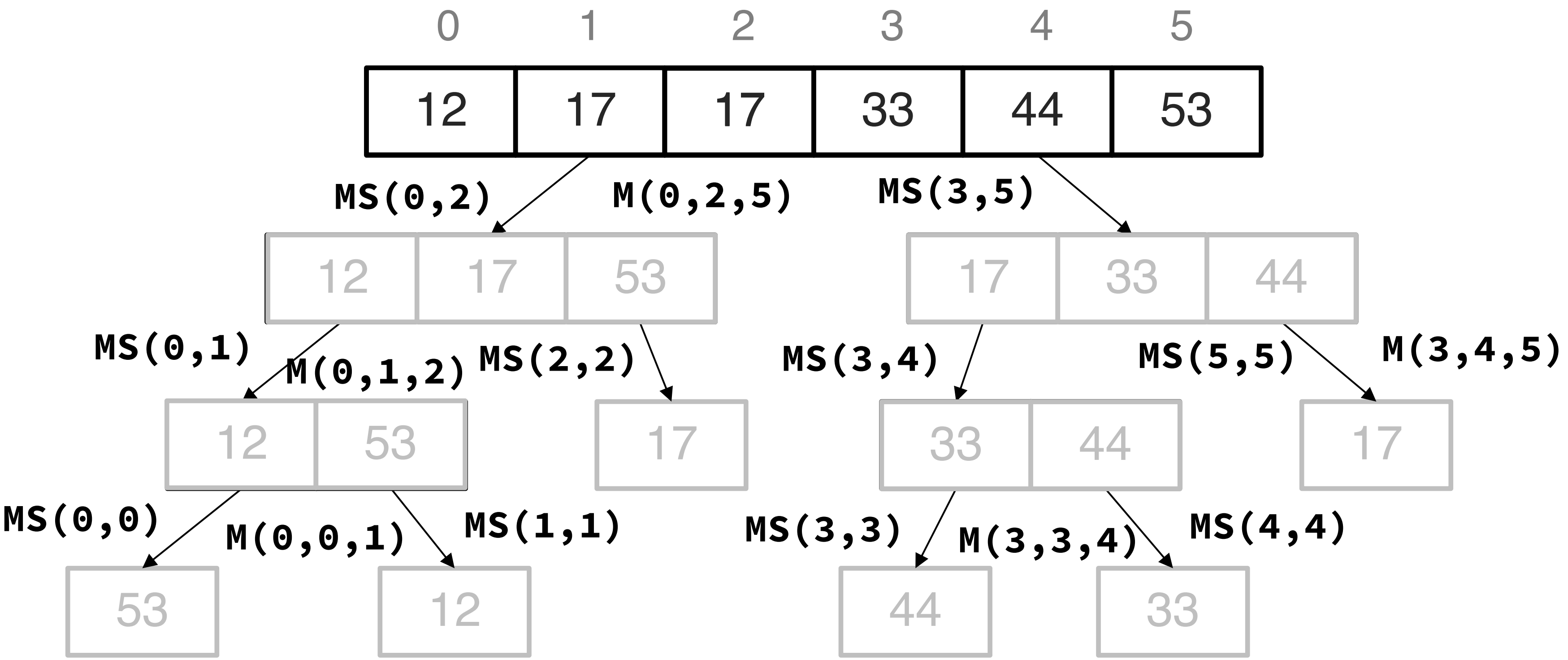
Beispiel: Merge Sort (V)

```
IF left<right THEN  //more than one element
  mid=floor((left+right)/2); // middle (rounded down)
  mergeSort(A,left,mid);     // sort left part
  mergeSort(A,mid+1,right);  // sort right part
  merge(A,left,mid,right);   // merge into one
```



Beispiel: Merge Sort (VI)

```
IF left<right THEN  //more than one element
  mid=floor((left+right)/2); // middle (rounded down)
  mergeSort(A,left,mid);     // sort left part
  mergeSort(A,mid+1,right);  // sort right part
  merge(A,left,mid,right);   // merge into one
```



Laufzeitanalyse: Rekursionsgleichungen

Laufzeitabschätzung Merge Sort

```
IF left<right THEN  //more than one element
  mid=floor((left+right)/2); // middle (rounded down)
  mergeSort(A,left,mid);     // sort left part
  mergeSort(A,mid+1,right);  // sort right part
  merge(A,left,mid,right);   // merge into one
```

Sei $T(n)$ die (maximale) Anzahl von Schritten für Arrays der Größe n:

$$T(n) \leq 2 \cdot T(n/2) + d + en \qquad \text{für } n \geq 2$$

zwei rekursive Aufrufe für jeweils halb so großes Array

(ignorieren hier zur Vereinfachung Runden)

`IF` + **`floor`**
(konstanter Aufwand)

`Merge`
(Aufwand $\mathcal{O}(\mathrm{n})$)

und $T(1) \leq d$,
weil dann **`left=right`**

Rekursion „manuell iterieren“

Bemerkung: Es gilt auch
$T(n) \geq \Omega(n \cdot \log n)$

Laufzeit Merge Sort
$\Theta(n \cdot \log n)$

$$
\begin{aligned}
T(n) &\leq 2 \cdot T(n/2) + cn \\
&\leq 2 \cdot (2 \cdot T(n/4) + cn/2) + cn \\
&\leq 2 \cdot (2 \cdot (2 \cdot T(n/8) + cn/4) + cn/2) + cn \\
&\;\;\vdots \\
&\leq 2 \cdot (2 \cdot (2 \cdots (2 \cdot T(1) + 2) \cdots + cn/4) + cn/2) + cn \\
&\leq \underbrace{2 \cdot (2 \cdot (2 \cdots (2 \cdot}_{\log_2 n - \mathrm{mal}} \quad c \quad + 2) \cdots + cn/4) + cn/2) + cn \\
&\leq 2^{\log_2 n} \cdot c + \log_2 n \cdot cn \quad = \mathcal{O}(n \log n)
\end{aligned}
$$

Allgemeiner Ansatz: Mastermethode

Allgemeine Form der Rekursionsgleichung:

$$T(n) = a \cdot T(n/b) + f(n), \qquad T(1) = \Theta(1)$$

mit $a \geq 1, b > 1$ und $f(n)$ eine asymptotisch positive Funktion

n/b müsste gerundet werden (hat aber keinen Einfluss auf asymptotisches Resultat)

Interpretation:

Problem wird in a Teilprobleme der Größe n/b aufgeteilt

Lösen jedes der a Teilprobleme benötigt Zeit $T(n/b)$

Funktion $f(n)$ umfasst Kosten für Aufteilen und Zusammenfügen

Mastertheorem

nach Cormen et al., Introduction to Algorithms

Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f(n)$ eine positive Funktion und $T(n)$ über den nicht-negativen ganzen Zahlen durch die Rekursionsgleichung

$$T(n) = aT(n/b) + f(n), \qquad\qquad T(1) = \Theta(1)$$

definiert, wobei wir n/b so interpretieren, dass damit entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ gemeint ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken:

1. Gilt $f(n) = O(n^{\log_b (a) - \epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$

2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$

3. Gilt $f(n) = \Omega(n^{\log_b (a) + \epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Interpretation (I)

entscheidend ist Verhältnis von $f(n)$ zu $n^{\log_b a}$:

1. Wenn $f(n)$ polynomiell kleiner als $n^{\log_b a}$, dann $T(n) = \Theta(n^{\log_b a})$
2. Wenn $f(n)$ und $n^{\log_b a}$ gleiche Größenordnung, dann $T(n) = \Theta(n^{\log_b a} \cdot \log n)$
3. Wenn $f(n)$ polynomiell größer als $n^{\log_b a}$ und $af(n/b) \leq cf(n)$, dann $T(n) = \Theta(f(n))$

Unterschied
polynomieller
Faktor n^{ε}

---

1. Gilt $f(n) = O(n^{\log_b(a) - \epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$
2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
3. Gilt $f(n) = \Omega(n^{\log_b(a) + \epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Interpretation (II)

„Regularität“ $af(n/b) \leq cf(n), c < 1$ in Fall 3

$$T(n) = a \cdot T(n/b) + f(n) = \underbrace{a \cdot \big(a \cdot T(n/b^2) + f(n/b)\big) + f(n)}$$

Aufwand $f(n)$ zum Teilen und Zusammenfügen für Größe n
dominiert (asymptotisch) Summe $af(n/b)$ aller Aufwände für Größe n/b

braucht man nur im dritten Fall für „große“ $f(n)$

1. Gilt $f(n) = O(n^{\log_b(a)-\epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$
2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
3. Gilt $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Beispiele Mastertheorem (I)

Merge Sort $T(n) = 2 \cdot T(n/2) + cn$

Fall 2

$$a = b = 2, \log_b a = 1$$
$$f(n) = \Theta(n) = \Theta(n^{\log_b a})$$

$$T(n) = \Theta\left(n^{\log_b a} \cdot \log_2 n\right) = \Theta(n \cdot \log_2 n)$$

$T(n) = a \cdot T(n/b) + f(n)$ mit $a \geq 1, b > 1$, $f(n)$ positiv

1. Gilt $f(n) = O(n^{\log_b (a)-\epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$

2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$

3. Gilt $f(n) = \Omega(n^{\log_b (a)+\epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Beispiele Mastertheorem (II)

$$T(n) = 4 \cdot T(n/3) + cn$$

Fall 1

$$a = 4, b = 3,\ \log_b a = 1.26\ldots, \varepsilon = 0.26\ldots$$
$$f(n) = \Theta(n) = \Theta(n^{\log_b(a)-\varepsilon})$$

$$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^{1.26\ldots})$$

$T(n) = a \cdot T(n/b) + f(n)$ mit $a \geq 1, b > 1$, $f(n)$ positiv

1. Gilt $f(n) = O(n^{\log_b(a)-\epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$
2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
3. Gilt $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Beispiele Mastertheorem (III)

$$T(n) = 3 \cdot T(n/3) + cn^2$$

Fall 3

$$a = 3, b = 3,\ \log_b a = 1, \varepsilon = 1$$

$$f(n) = \Theta(n^2) = \Theta(n^{\log_b(a)+\varepsilon})$$

$$3f(n/3) = cn^2/3 \leq \tfrac{1}{3} \cdot f(n)$$

$$\downarrow$$

$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

$T(n) = a \cdot T(n/b) + f(n)$ mit $a \geq 1, b > 1,\ f(n)$ positiv

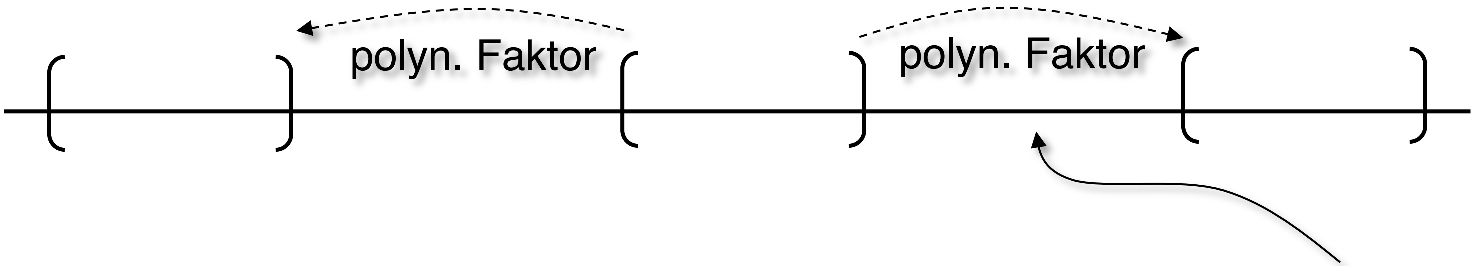
1. Gilt $f(n) = O(n^{\log_b(a)-\epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$

2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$

3. Gilt $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Grenzen des Mastertheorems

$f(n) = O(n^{\log_b(a)-\varepsilon})$ $\qquad$ $f(n) = \Theta(n^{\log_b a})$ $\qquad$ $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$



$T(n) = 2 \cdot T(n/2) + n \log n$ $\qquad$ $a = b = 2, \log_b a = 1, f(n) = n \log n$

(iterativ: $\boldsymbol{T(n) = \Theta(n \cdot \log^2 n)}$) $\qquad$ also $\boldsymbol{f(n) \notin \Theta(n)}$ und $\boldsymbol{f(n) \notin \Omega(n^{1+\varepsilon})}$

1. Gilt $f(n) = \mathrm{O}(n^{\log_b(a)-\epsilon})$ für eine Konstante $\epsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$

2. Gilt $f(n) = \Theta(n^{\log_b a})$, dann gilt $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$

3. Gilt $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und hinreichend große n, dann ist $T(n) = \Theta(f(n))$

Wieso gilt für die Laufzeit bei Merge Sort $T(n) = \Omega(n \cdot \log n)$?

Lösen Sie folgende Rekursionsgleichung

$$T(n) = 4\,T(n/4) + n^2 \log n$$

mit Hilfe des Mastertheorems.