



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

# Algorithmen und Datenstrukturen



SYSTEMS

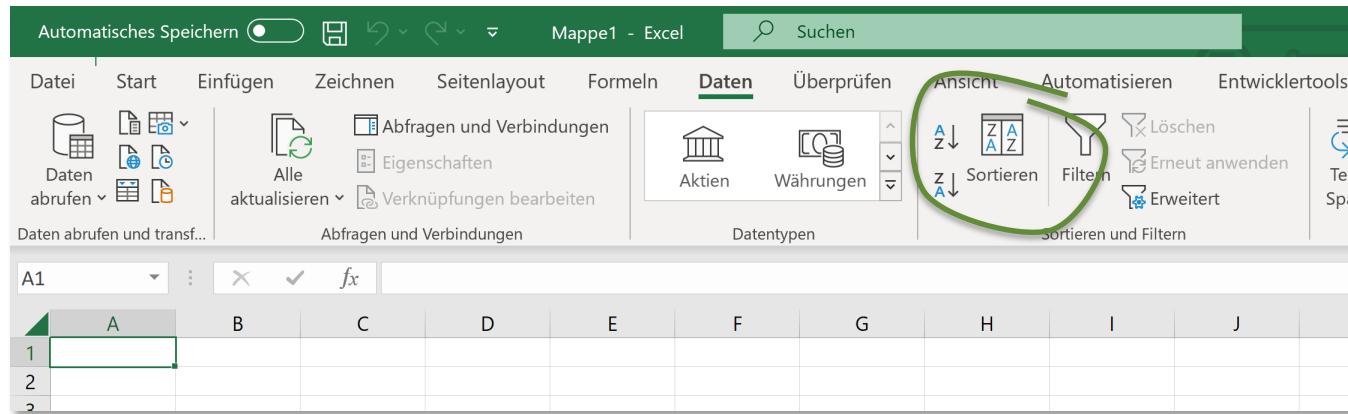
Stefan Roth, SS 2025

02

Sortieren

# Das Sortierproblem

# Sortierproblem in der Praxis



The screenshot shows a library search interface with the following elements:

- Header: Bücher & mehr (9023), Aufsätze & mehr (104294), Bibliothekskonto
- Section: Bücher, Zeitschriften und mehr
- Text: Treffer 1 - 25 von 9023, Suchdauer: 0.63s
- Navigation: 1 2 3 4 5 6 7 8 9 10 11 weiter [361]
- Left sidebar: Ergebnis einschränken (Filter), Standort (Location):
  - Online-Ressourcen (4523)
  - ULB Darmstadt - Stadtmitte (3779)
  - Handapparate Stadtmitte (848)
  - ULB Darmstadt - Lichtwiese (545)
  - Handapparate Lichtwiese (156)
- Results:
  - 1 Ebook Volltext: Evolutionäre Algorithmen - Genetische Algorithmen zur Musikkomposition by Evelyn Elsner, published by Hochschule Hannover, 2021.
  - 2 Ebook: Algorithmen im Alltag
- Right sidebar: Sortieren (Sort) dropdown menu:
  - Relevanz (selected)
  - Nach Datum, absteigend
  - Nach Datum, aufsteigend
  - Verfasser
  - Titel

# Sortierproblem

**Gegeben:** Folge von Objekten

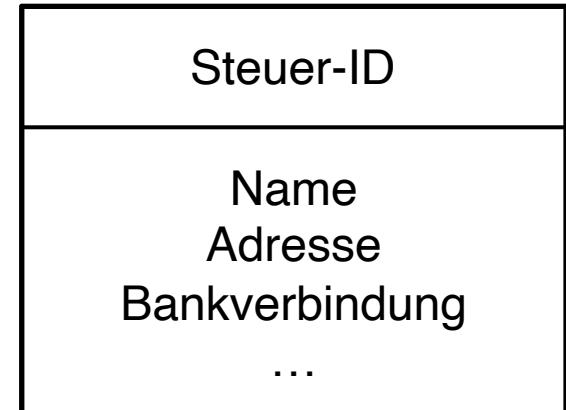
**Gesucht:** Sortierung der Objekte  
(gemäß ausgewiesenen Schlüsselwerts)



Objekt

Wert, nach dem Objekt sortiert wird

Beispiel



# Schlüsselproblem

Schlüssel(werte) müssen nicht eindeutig sein,

**aber müssen „sortierbar“ sein**



Objekt

Wert, nach dem Objekt sortiert wird

Annahme im Folgenden:

Es gibt eine totale Ordnung  
 $\leq$  auf der Menge  $M$  aller  
möglichen Schlüsselwerte

# Totale Ordnung

Beispiel: lexikographische Ordnung auf Strings

Sei  $M$  eine nicht leere Menge und  $\leq \subseteq M \times M$  eine binäre Relation auf  $M$

Die Relation  $\leq$  auf  $M$  ist genau dann eine totale Ordnung, wenn gilt:

Reflexivität:  $\forall x \in M: x \leq x$

Transitivität:  $\forall x, y, z \in M: x \leq y \wedge y \leq z \Rightarrow x \leq z$

Antisymmetrie:  $\forall x, y \in M: x \leq y \wedge y \leq x \Rightarrow x = y$

Totalität:  $\forall x, y \in M: x \leq y \vee y \leq x$

(ohne Totalität:  
partielle Ordnung)

Bemerkung: Totale Ordnung  $\leq$  impliziert Relation  $>$  durch  $x > y: \Leftrightarrow x \not\leq y$

# Darstellung in diesem Abschnitt

Wir betrachten nur Schlüssel(werte) ohne Satellitendaten,  
in Beispielen meistens durch Zahlen gegeben

Eingabe der Objekte bzw. Schlüsselwerte in Form eines Arrays A:

	0	1	2	3	4	5	6	7	8
A	53	12	17	44	33	25	17	4	76

Lese-/Schreibzugriff per Index in konstanter Zeit:

$y = A[2]$  weist  $y$  den Wert 17 zu

$A[4] = 99$  überschreibt 33 mit 99

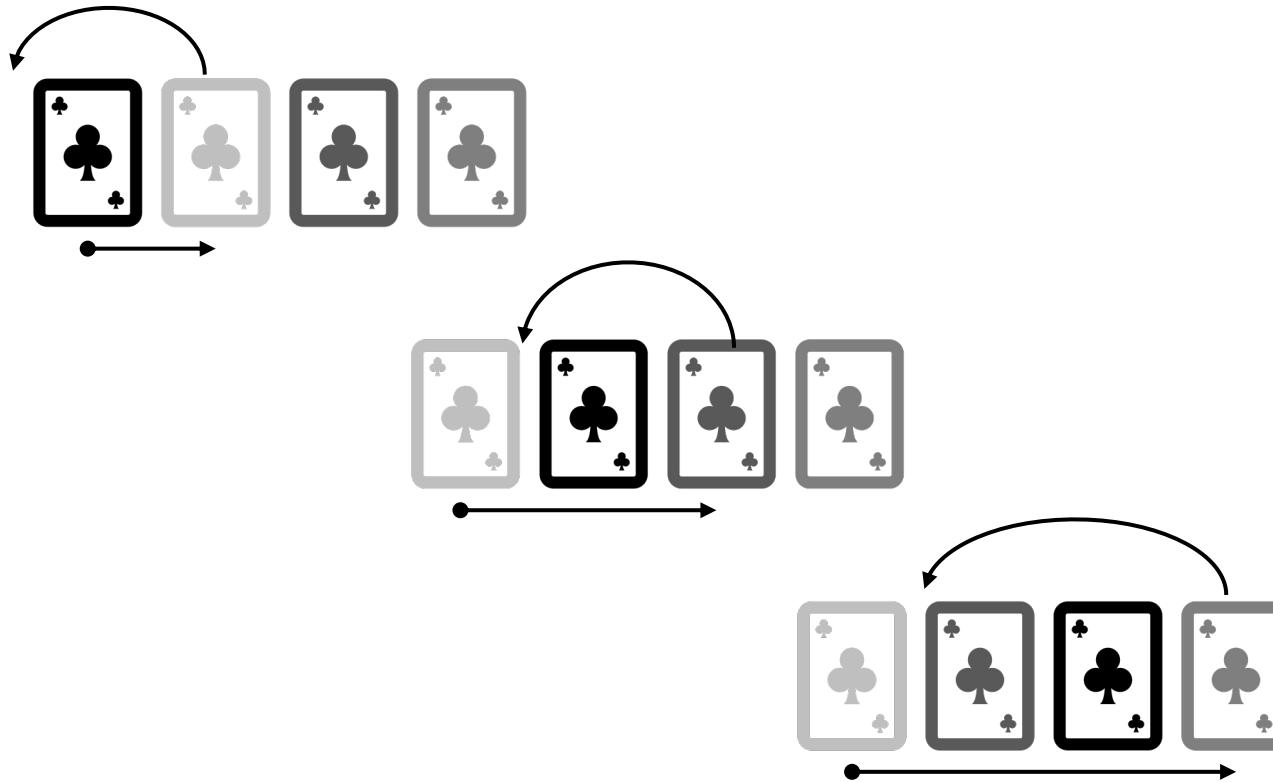
**A.length** beschreibt (fixe) Länge des Arrays, im Beispiel 9

**A[i...j]** beschreibt Teil-Array von Index  $i$  bis  $j$ ,  $0 \leq i \leq j \leq A.length - 1$

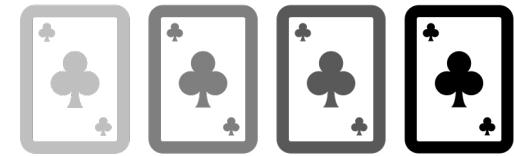
# Insertion Sort

# Idee: Karten umsortieren

(hell nach dunkel)



Gehe von links nach rechts durch  
und sortiere aktuelle Karte richtig nach links ein



# Algorithmus: Insertion Sort

Durch  $A[j] < key$   
wohldefiniert

```
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;
```

Wir beginnen mit **i=1**, aber erstes Element ist **A[0]**

Short Circuit Evaluation (wie in Java):

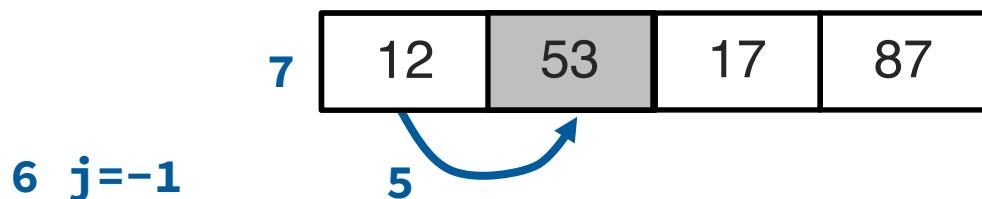
Wenn erste AND-Bedingung **false**, wird zweite Bedingung nicht ausgewertet

# Beispiel: Insertion Sort (I)

```
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;
```

Beispiel: FOR-Schleife  $i=1$       2 key=12      3  $j=0$

WHILE-Schleife  $j=0$        $A[j]=53 > key=12$



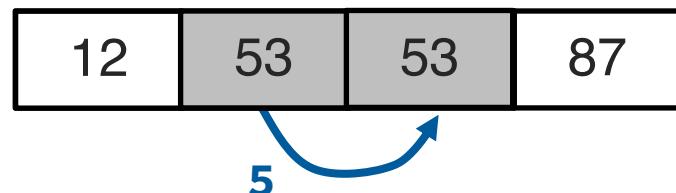
1. Fall:  
linker Rand erreicht

# Beispiel: Insertion Sort (II)

```
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;
```

Beispiel: FOR-Schleife  $i=2$       2 key=17      3  $j=1$

WHILE-Schleife  $j=1$      $A[j]=53 > key=17$



6  $j=0$

2. Fall:  
Einfügeposition erreicht

# Beispiel: Insertion Sort (III)

```
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;
```

Beispiel: FOR-Schleife  $i=2$       2 key=17      3  $j=1$

WHILE-Schleife  $j=0$        $A[j]=12 < \text{key}=17$

12	17	53	87
6 $j=0$	7		

2. Fall:

Einfügeposition erreicht

# Beispiel: Insertion Sort (IV)

```
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;
```

Beispiel: FOR-Schleife  $i=3$       2 key=87      3  $j=2$

WHILE-Schleife  $j=2$        $A[j]=53$  < key=87

12	17	53	87
7			

3. Fall:  
Element bleibt

# Terminierung

```
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;
```

Jede Ausführung der **WHILE**-Schleife erniedrigt  $j < n$  in jeder Iteration um 1 und bricht ab, wenn  $j < 0 \rightarrow$  terminiert also immer

**FOR**-Schleife wird nur endlich oft durchlaufen

# Korrektheit (I)

```
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;
```

Schleifeninvariante (der **FOR**-Schleife):

Bei jedem Eintritt für Zählerwert **i** (und nach letzter Ausführung) entsprechen die aktuellen Einträge in **A[0],...,A[i-1]** den sortierten ursprünglichen Eingabewerten **a[0],...,a[i-1]**. Ferner gilt **A[i]=a[i],..., A[n-1]=a[n-1]**.

# Beweis per vollständiger Induktion

Klassische **mathematische Beweismethode** für Aussagen  $A(n)$ , die für **alle natürlichen Zahlen**  $n \in \mathbb{N}$  gelten, die größer oder gleich einem **bestimmten Startwert**  $n_0$  sind.  $n_0$  ist häufig 0 oder 1.

Behauptung: Aussage  $A(n)$  gilt  $\forall n \in \mathbb{N}, n \geq n_0 \in \mathbb{N}$ .

Beweisstruktur:

**Induktionsanfang:** Zeige, dass Aussage  $A(n_0)$  für Startwert  $n_0$  gilt.

**Induktionsschritt:** Zeige, dass wenn  $A(n)$  für ein beliebiges  $n \in \mathbb{N}, n \geq n_0$  gilt (*Induktionsannahme*), auch  $A(n + 1)$  gelten muss (*Induktionsbehauptung*).

**Induktionsschluss:**\* Aus Induktionsanfang und Induktionsschritt folgt die Behauptung.

\* Wird der Vereinfachung halber oft weggelassen

# Beweis per vollständiger Induktion – Beispiel I

Gauß'sche Summenformel

Behauptung:

Aussage  $A(n)$ :  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$  gilt  $\forall n \in \mathbb{N}$ .

Beweis:

**Induktionsanfang:**

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

**Induktionsschritt:**

$n \rightarrow n + 1$ :

$$\begin{aligned} \sum_{i=1}^{n+1} i &= \sum_{i=1}^n i + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} \\ &= \frac{(n+2)(n+1)}{2} = \frac{(n+1)((n+1)+1)}{2} \end{aligned}$$

■

# Beweis per vollständiger Induktion – Beispiel II

*Bernoulli'sche Ungleichung*

Behauptung:

Aussage  $A(n)$ :  $(1 + x)^n \geq 1 + nx$   
gilt  $\forall n \in \mathbb{N}_0$  und  $\forall x \in \mathbb{R}, x > -1$ .

Beweis:

**Induktionsanfang:**  $n = 0$ :  $(1 + x)^n = (1 + x)^0 = 1 = 1 + 0x = 1 + nx$

**Induktionsschritt:**  $n \rightarrow n + 1$ :

$$\begin{aligned}
 (1 + x)^{(n+1)} &= (1 + x)^n \cdot (1 + x) \geq (1 + nx) \cdot (1 + x) = 1 + nx + x + nx^2 \\
 &= 1 + (n + 1)x + nx^2 \geq 1 + (n + 1)x \\
 &\quad \text{Induktionsannahme} \\
 &\quad \geq 0
 \end{aligned}$$

■

# Korrektheit (II)

Beweis per Induktion:

Induktionsbasis  $i=1$ : Beim ersten Eintritt ist  $A[0]=a[0]$  und „sortiert“. Ferner gilt noch  $A[1]=a[1], \dots, A[n-1]=a[n-1]$ .

Bei jedem Eintritt für Zählerwert  $i$  (und nach letzter Ausführung) entsprechen die aktuellen Einträge in  $A[0], \dots, A[i-1]$  den sortierten ursprünglichen Eingabewerten  $a[0], \dots, a[i-1]$ . Ferner gilt  $A[i]=a[i], \dots, A[n-1]=a[n-1]$ .

# Korrektheit (III)

**Beweis per Induktion:** Induktionsschritt von  $i-1$  auf  $i$ :

Vor der ( $i-1$ )-ten Ausführung galt Schleifeninvariante nach Voraussetzung. Inbesondere war  $A[0], \dots, A[i-2]$  sortierte Version von  $a[0], \dots, a[i-2]$  und  $A[i-1] = a[i-1], \dots, A[n-1] = a[n-1]$

Durch die **WHILE**-Schleife wurde  $A[i-1] = a[i-1]$  nach links eingesortiert und größere Elemente von  $A[0], \dots, A[i-2]$  um jeweils eine Position nach rechts verschoben. Elemente  $A[i], \dots, A[n-1]$  wurden nicht geändert

Also gilt Invariante auch für  $i$

Bei jedem Eintritt für Zählerwert  $i$  (und nach letzter Ausführung) entsprechen die aktuellen Einträge in  $A[0], \dots, A[i-1]$  den sortierten ursprünglichen Eingabewerten  $a[0], \dots, a[i-1]$ . Ferner gilt  $A[i] = a[i], \dots, A[n-1] = a[n-1]$ .

# Korrektheit (IV)

Aus Schleifeninvariante folgt für letzte Ausführung  
(also quasi vor gedanklichem Eintritt der Schleife für  $i=n$ ):

$A[0], \dots, A[n-1]$  ist sortierte Version von  $a[0], \dots, a[n-1]$

und somit am Ende das Array sortiert

Bei jedem Eintritt für Zählerwert  $i$  (und nach letzter Ausführung) entsprechen die aktuellen Einträge in  $A[0], \dots, A[i-1]$  den sortierten ursprünglichen Eingabewerten  $a[0], \dots, a[i-1]$ . Ferner gilt  $A[i]=a[i], \dots, A[n-1]=a[n-1]$ .



Wozu brauchen Sie (intuitiv) beim Sortieren die vier Eigenschaften einer totalen Ordnung?



Überlegen Sie sich, dass Insertion Sort **stabil** ist, d.h. die Reihenfolge von Objekten mit gleichem Schlüssel bleibt erhalten.

# Laufzeitanalysen: $\mathcal{O}$ -Notation

# Laufzeitanalyse

Wieviele Schritte macht Algorithmus  
in Abhängigkeit von der Eingabekomplexität?

meistens: schlechtester Fall über  
alle Eingaben gleicher Komplexität

fasst alle Eingaben  
ähnlicher Komplexität zusammen

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

Beispiel:  $n$  Anzahl zu  
sortierender Zahlen

Maximum über alle Eingaben  
x der Komplexität  $n$

(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  $c_i$

# 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 A
2     key=A[i];
3     j=i-1; // search for insertion point b/w A[j+1] and A[i]
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;
```

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 A
2     key=A[i];
3     j=i-1; // search for insertion point b/w A[j+1] & A[i]
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;
```

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:

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

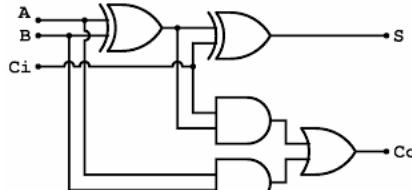
Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile  $i$  hat Aufwand  $c_i$

# Kosten für individuelle Schritte?

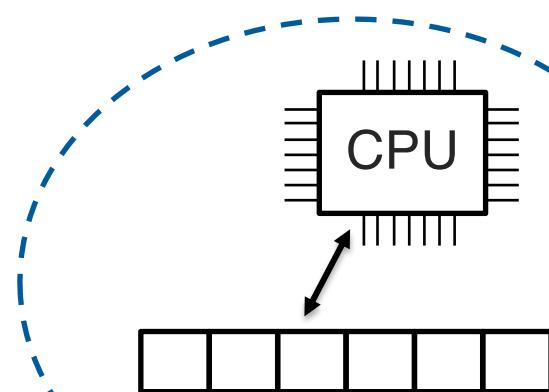
Wie teuer ist z.B. Zuweisung  $A[j+1]=A[j]$  in Zeile 5, also was ist  $c_5$ ?

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

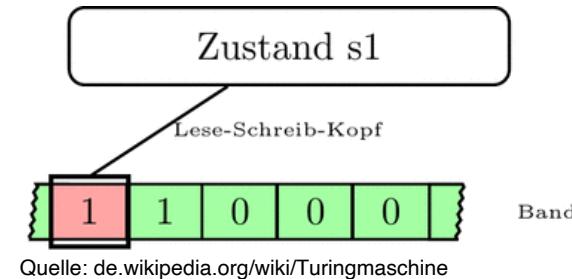


Quelle: ke.tu-darmstadt.de

Schaltkreis



RAM



Quelle: de.wikipedia.org/wiki/Turingmaschine

Turingmaschine

nehmen üblicherweise an, dass elementare Operationen  
(Zuweisung, Vergleich,...) in einem Schritt möglich →  $c_5 = 1$

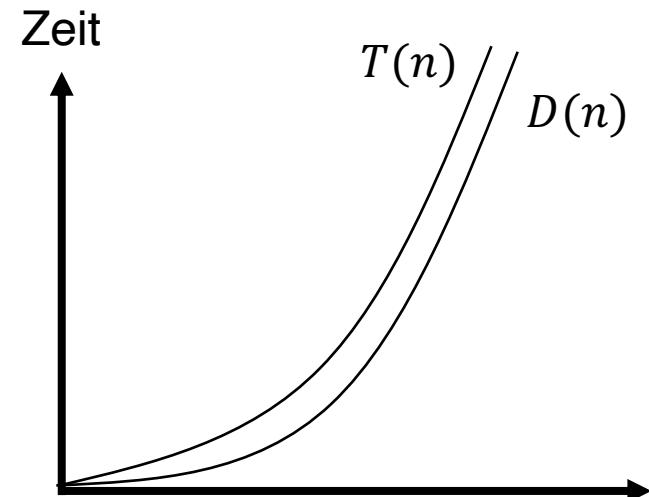
# Asymptotische Vereinfachung (I)

Gesamtlauzeit 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) :

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



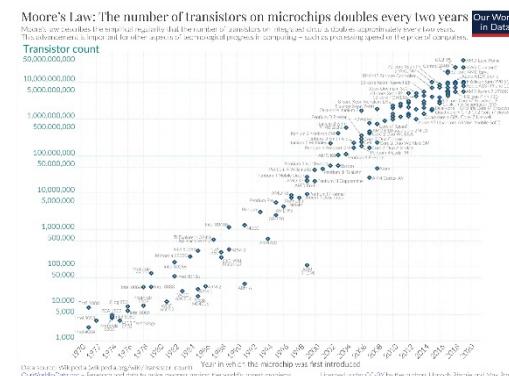
Zum Vergleich (nur dominanter Term) :

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

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/Moore%27s\\_Gesetz](https://de.wikipedia.org/wiki/Moore%27s_Gesetz)

# $\Theta$ -Notation / Landau-Symbole

Paul Bachmann  
Edmund Landau  
ca. 1900

Funktionen  $f, g: \mathbb{N} \rightarrow \mathbb{R}_{>0}$

Eingabekomplexität

Laufzeit

$$\Theta(g) = \{f : \exists c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}$$

Funktion  $f$

Positive Konstanten

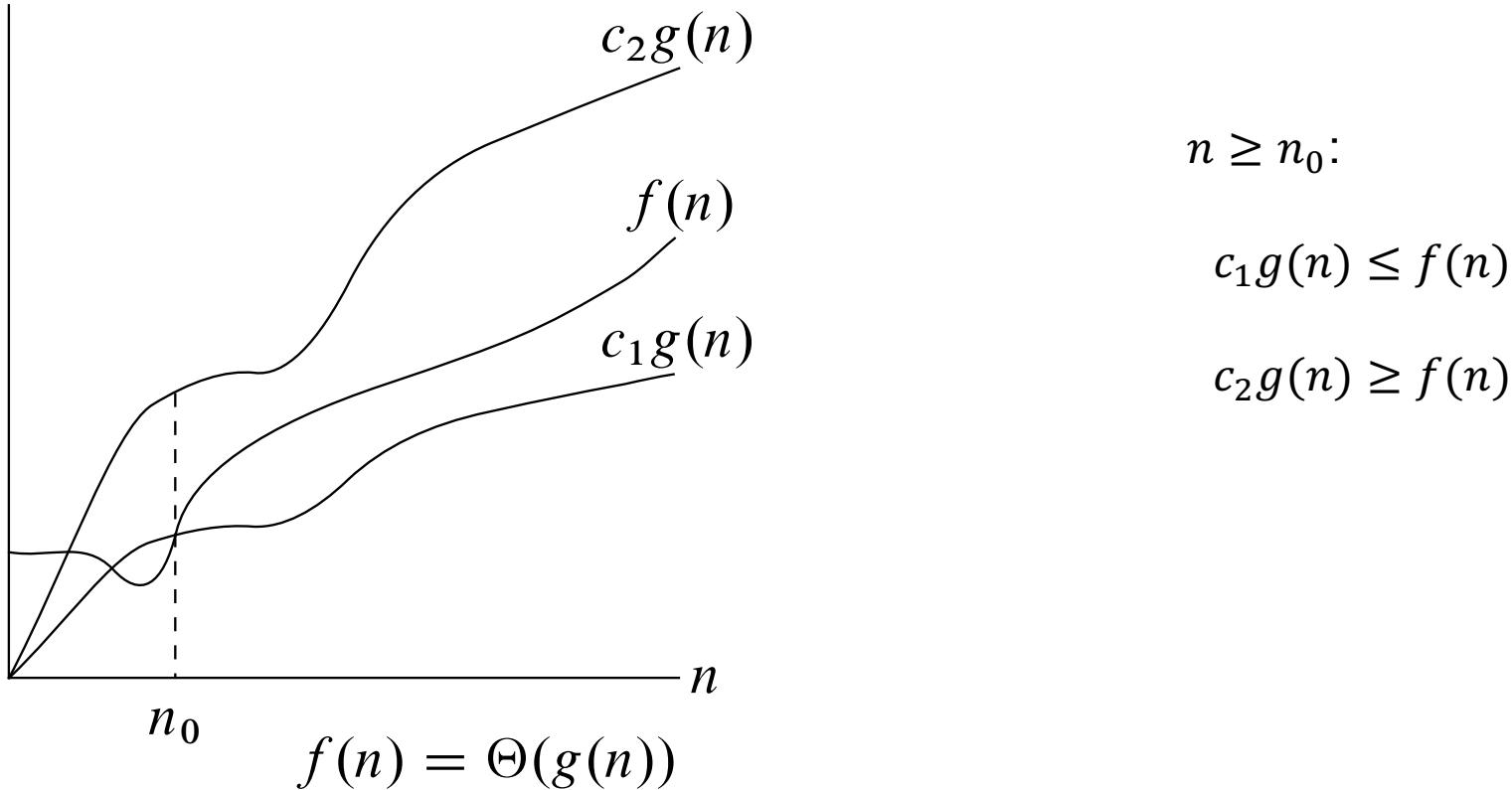
Für alle  $n$  größer gleich  $n_0$

$f(n)$  wird von  $c_1g(n)$  und  $c_2g(n)$  für hinreichend große  $n$  eingeschlossen

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

# Veranschaulichung $\Theta$ -Notation

Quelle: Cormen et al. Introduction to Algorithms



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

$\Theta$ -Notation beschränkt eine Funktion asymptotisch von oben und unten

# Beispiel: Laufzeit Insertion Sort in $\Theta$ -Notation (I)

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

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

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

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

# Beispiel: Laufzeit Insertion Sort in $\Theta$ -Notation (II)

$$T(n) = n + 4 \cdot (n - 1) + 3 \cdot \frac{n(n-1)}{2} = \Theta(n^2) \quad \begin{array}{l} \text{für } c_1 = \frac{3}{2}, \\ c_2 = 7, n_0 = 2 \end{array}$$

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

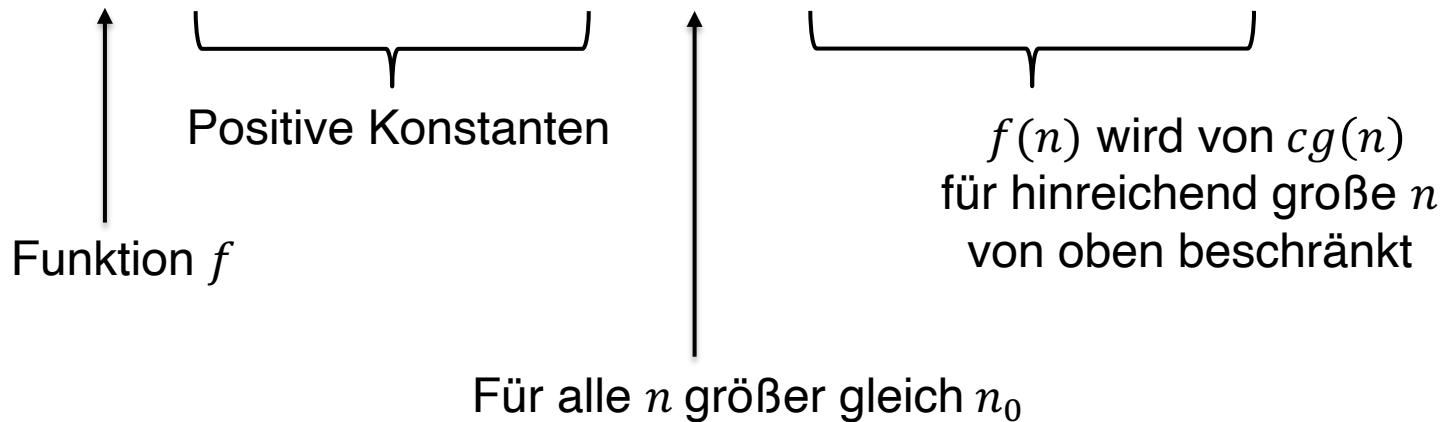
$$\begin{aligned} T(n) &\leq n + 4 \cdot n + 2 \cdot n(n - 1) \\ &\leq 5 \cdot n + 2 \cdot n^2 \\ &\leq 5 \cdot n^2 + 2 \cdot n^2 \\ &\leq 7 \cdot n^2 \end{aligned}$$

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

# $\mathcal{O}$ -Notation

Obere asymptotische Schranke

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

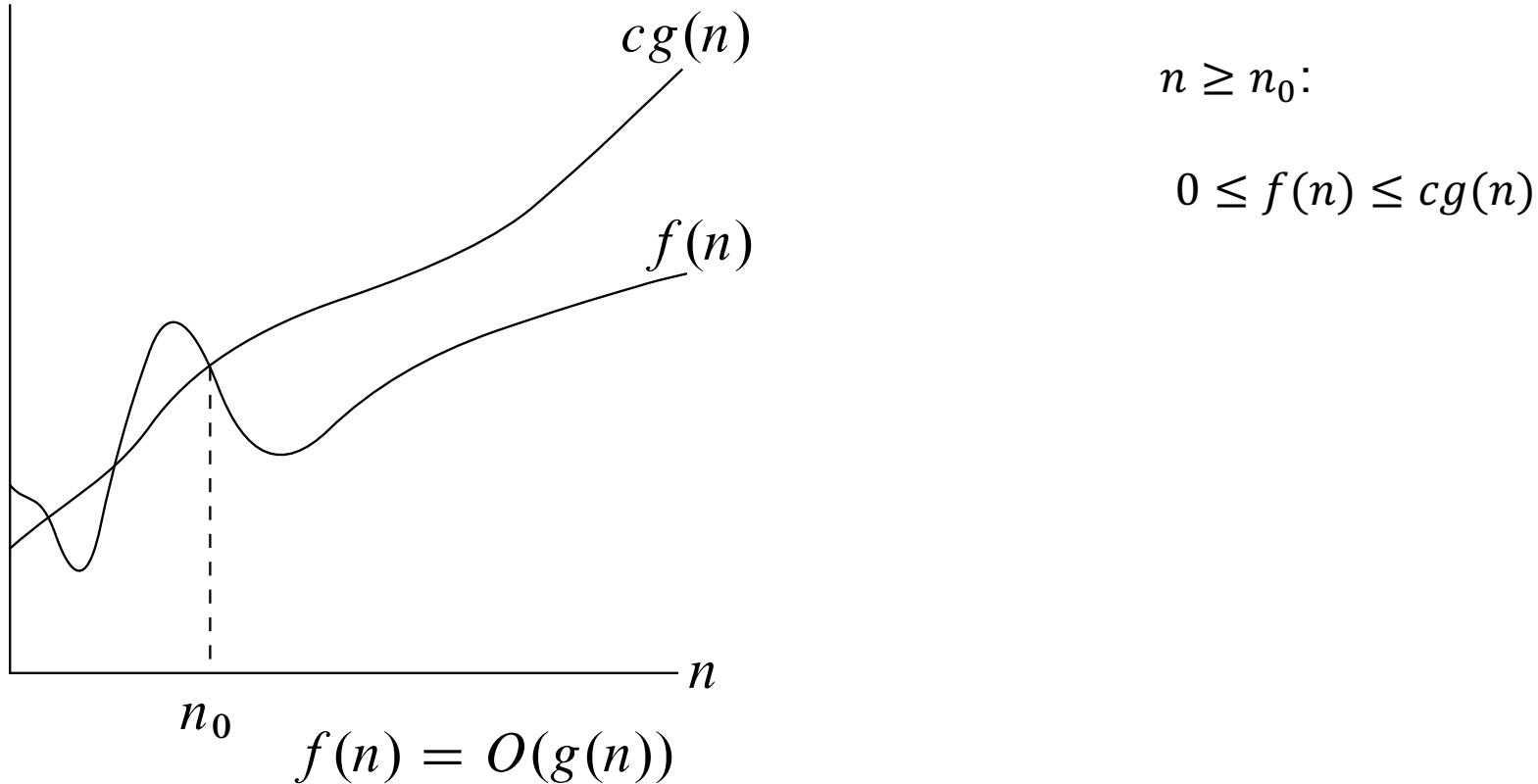


Sprechweise:  $f$  wächst höchstens so schnell wie  $g$

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

# Veranschaulichung der $\mathcal{O}$ -Notation

Quelle: Cormen et al. Introduction to Algorithms



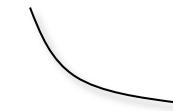
Beachte:  $\Theta(g(n)) \subseteq \mathcal{O}(g(n))$  und somit  $f(n) = \Theta(g) \Rightarrow f(n) = \mathcal{O}(g)$

# $\mathcal{O}$ -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  $f_1 + f_2 = \mathcal{O}(\max\{g_1, g_2\})$

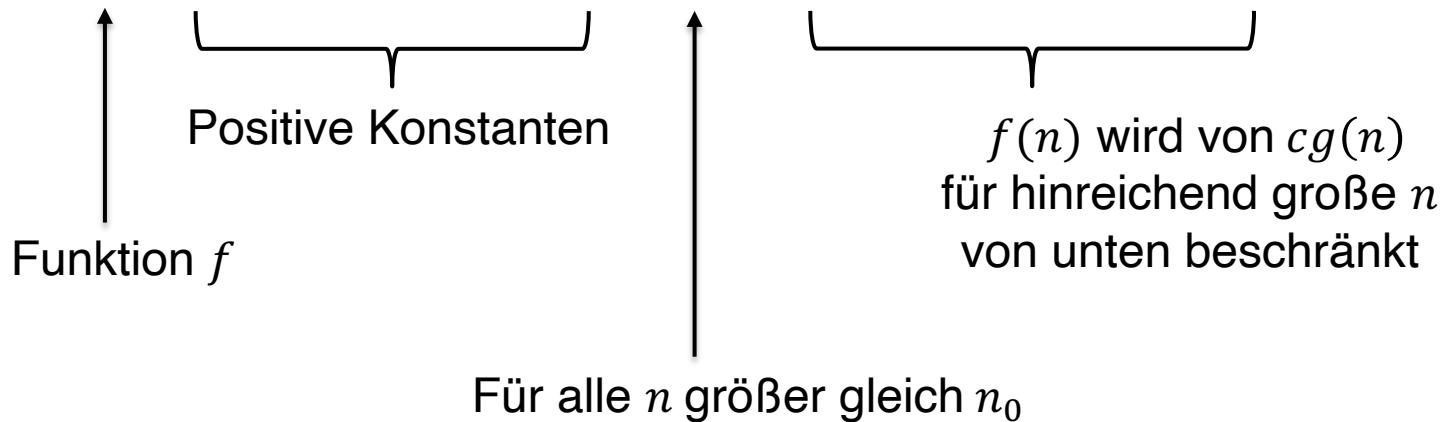
punktweise

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

# $\Omega$ -Notation

Untere asymptotische Schranke

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

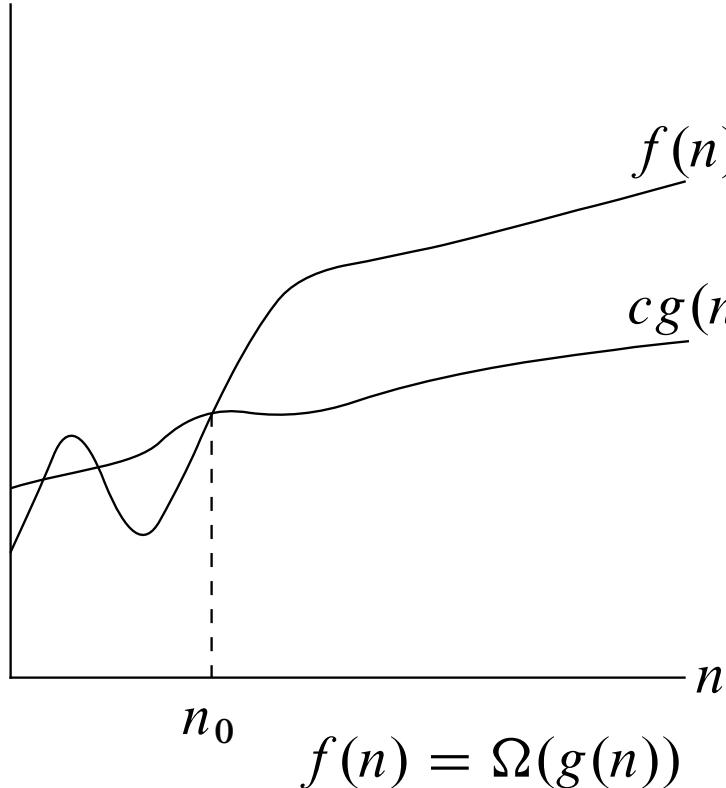


Sprechweise:  $f$  wächst mindestens so schnell wie  $g$

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

# Veranschaulichung der $\Omega$ -Notation

Quelle: Cormen et al. Introduction to Algorithms



$$n \geq n_0:$$

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

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



Überlegen Sie sich die Rechenregeln für  $\Omega$  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 $\Theta$

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

$$\begin{aligned} f(n) &= \Theta(g(n)) \\ \text{genau dann, wenn} \\ f(n) &= \mathcal{O}(g(n)) \text{ und } f(n) = \Omega(g(n)) \end{aligned}$$

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

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

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

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

# 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  $\in$ ,  $\subseteq$ ):

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

$$\in \quad / \quad \subseteq \quad \subseteq$$

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

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

```
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;
```

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 $\Theta$ bei Insertion Sort (II)

```
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;
```

zur Erinnerung: (Worst-Case-)Laufzeit  $T(n) = \max \{ \text{Anzahl Schritte f\"ur } x \}$

F\"ur untere Schranke muss man „nur“  
eine schlechte bzw. die schlechteste Eingabe  $x$  finden

Dann gilt  $T(n) \geq \text{Anzahl Schritte f\"ur schlechtes } x$

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

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

```
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;
```

„Schlechte“ Eingabe:

A	$n$	$n - 1$	$n - 2$		...	...		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)
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;
```

„gute“ Eingaben bereits vorsortiert, extremes Beispiel:

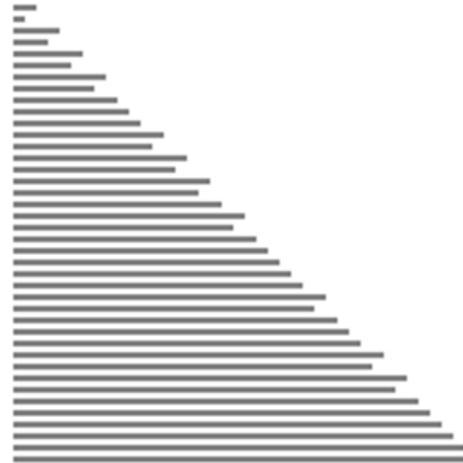
A	1	2	3		...	...		$n - 2$	$n - 1$
---	---	---	---	--	-----	-----	--	---------	---------

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

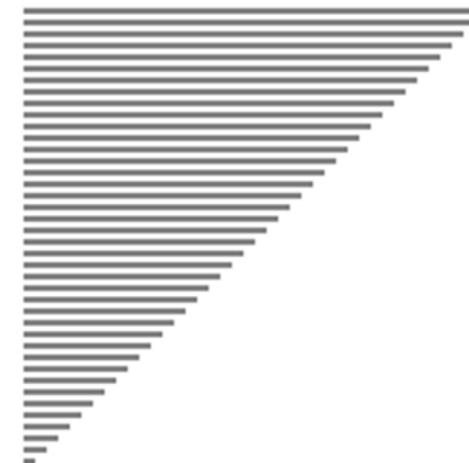
Insgesamt macht Algorithmus für dieses **A** also  $\Theta(n)$  Schritte

# Laufzeit Insertion Sort (I)

„guter“ Fall  
(fast vorsortiertes Array)



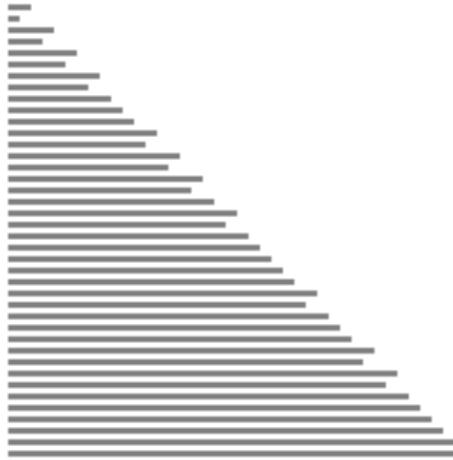
„schlechter“ Fall  
(invertiertes, vorsortiertes Array)



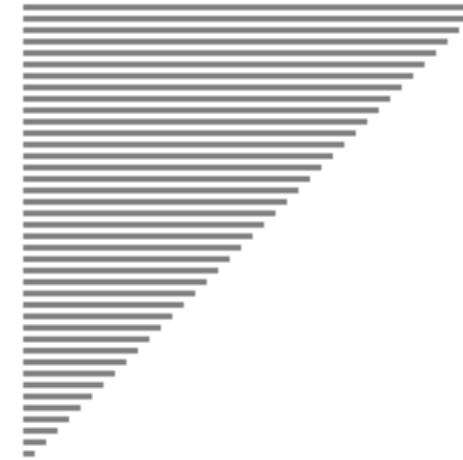
Quelle: <https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/insertion-sort>

# 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	
$\Theta(k^n)$	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 $\omega$ -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)
1 FOR i=A.length-1 DOWNT0 0 DO
2   FOR j=0 TO i-1 DO
3     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?



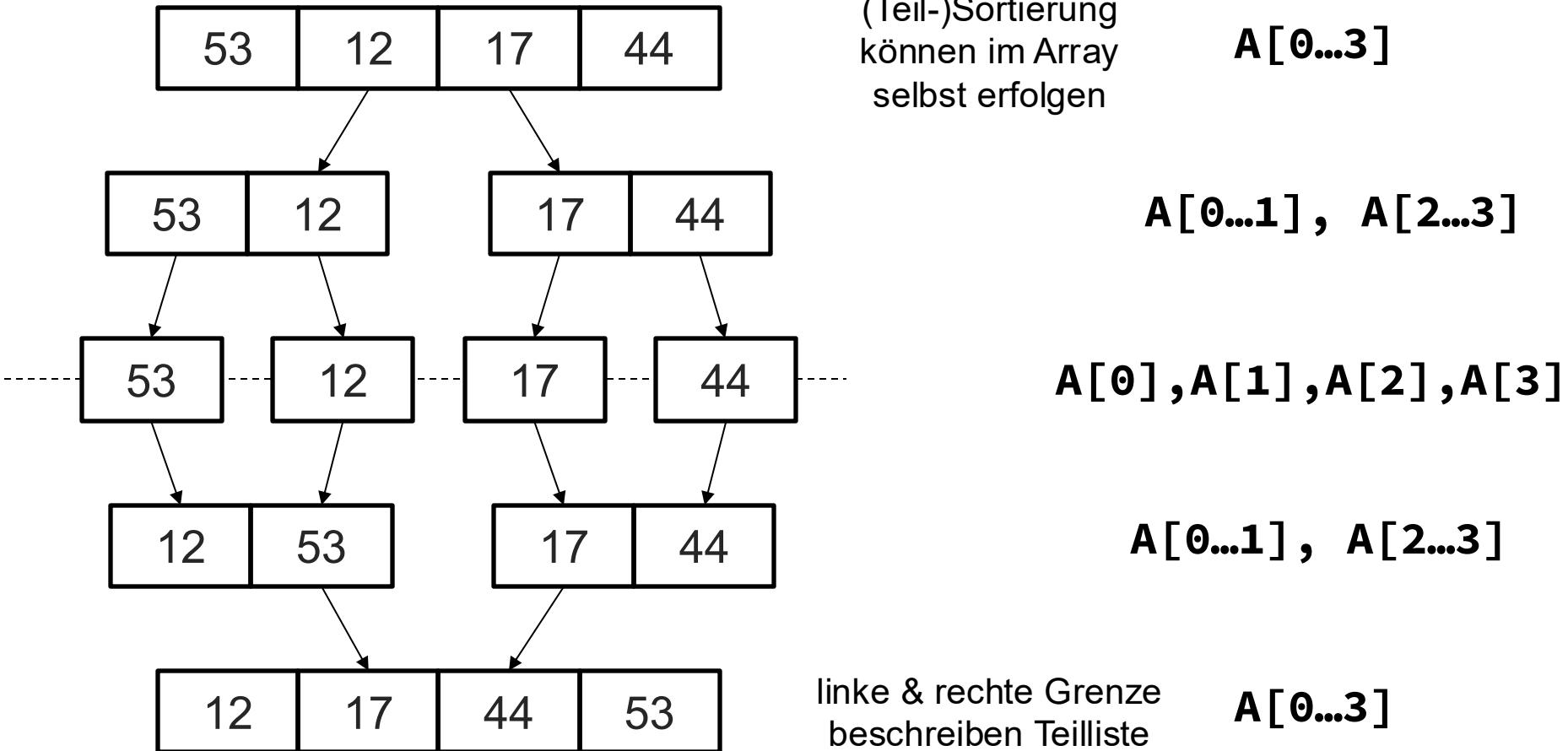
Wie verhält er sich im Vergleich zu Insertion Sort?

# Merge Sort

# Idee

## Divide & Conquer (& Combine)

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



# 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
```

```
1 IF left<right THEN //more than one element
2   mid=floor((left+right)/2); // middle (rounded down)
3   mergeSort(A, left, mid);    // sort left part
4   mergeSort(A, mid+1, right); // sort right part
5   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**

# 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

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
```

# **Laufzeitanalyse: Rekursionsgleichungen**

# Laufzeitabschätzung Merge Sort

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

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

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

zwei rekursive Aufrufe für  
jeweils halb so großes Array

(ignorieren hier zur  
Vereinfachung Runden)

**IF + floor**  
(konstanter Aufwand)

**Merge**  
(Aufwand  $\mathcal{O}(n)$ )

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

# Rekursion „manuell iterieren“

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

$$T(n) \leq 2 \cdot T(n/2) + cn$$

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

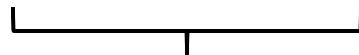
$$\leq 2 \cdot (2 \cdot T(n/4) + cn/2) + cn$$

$$\leq 2 \cdot (2 \cdot (2 \cdot T(n/8) + cn/4) + cn/2) + cn$$

⋮

$$\leq 2 \cdot (2 \cdot (2 \cdots (2 \cdot T(1) + 2) \cdots + cn/4) + cn/2) + cn$$

$$\leq 2 \cdot (2 \cdot (2 \cdots (2 \cdot c + 2) \cdots + cn/4) + cn/2) + cn$$



$\log_2 n - \text{mal}$

$$\leq 2^{\log_2 n} \cdot c + \log_2 n \cdot cn = \mathcal{O}(n \log n)$$

# Allgemeiner Ansatz: Mastermethode

## Allgemeine Form der Rekursionsgleichung:

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

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

### Interpretation:

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

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

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

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  $n^\varepsilon$

1. Gilt  $f(n) = O(n^{\log_b(a)-\varepsilon})$  für eine Konstante  $\varepsilon > 0$ , 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  $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$  für eine Konstante  $\varepsilon > 0$  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

$$T(n) = a \cdot T(n/b) + f(n) = a \cdot (a \cdot T(n/b^2) + f(n/b)) + f(n)$$



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

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

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

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

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

$$\begin{aligned} a &= 4, b = 3, \log_b a = 1.26 \dots, \varepsilon = 0.26 \dots \\ f(n) &= \Theta(n) = \Theta(n^{\log_b(a)-\varepsilon}) \end{aligned}$$

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{1.26\dots})$$

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

1. Gilt  $f(n) = O(n^{\log_b(a)-\varepsilon})$  für eine Konstante  $\varepsilon > 0$ , 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  $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$  für eine Konstante  $\varepsilon > 0$  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

$$\begin{aligned} & a = 3, b = 3, \log_b a = 1, \varepsilon = 1 \\ & f(n) = \Theta(n^2) = \Theta(n^{\log_b(a)+\varepsilon}) \\ & 3f(n/3) = cn^2/3 \leq \frac{1}{3} \cdot f(n) \end{aligned}$$

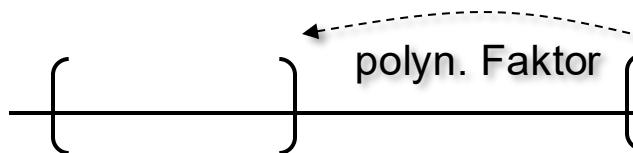
$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

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

1. Gilt  $f(n) = O(n^{\log_b(a)-\varepsilon})$  für eine Konstante  $\varepsilon > 0$ , 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  $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$  für eine Konstante  $\varepsilon > 0$  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

$$f(n) = O(n^{\log_b(a)-\varepsilon})$$



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

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

$$T(n) = 2 \cdot T(n/2) + n \log n$$

(iterativ:  $T(n) = \Theta(n \cdot \log^2 n)$ )

$$a = b = 2, \log_b a = 1, f(n) = n \log n$$

also  $f(n) \notin \Theta(n)$  und  $f(n) \notin \Omega(n^{1+\varepsilon})$

1. Gilt  $f(n) = O(n^{\log_b(a)-\varepsilon})$  für eine Konstante  $\varepsilon > 0$ , 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  $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$  für eine Konstante  $\varepsilon > 0$  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  $T(n) = \Omega(n \cdot \log n)$ ?



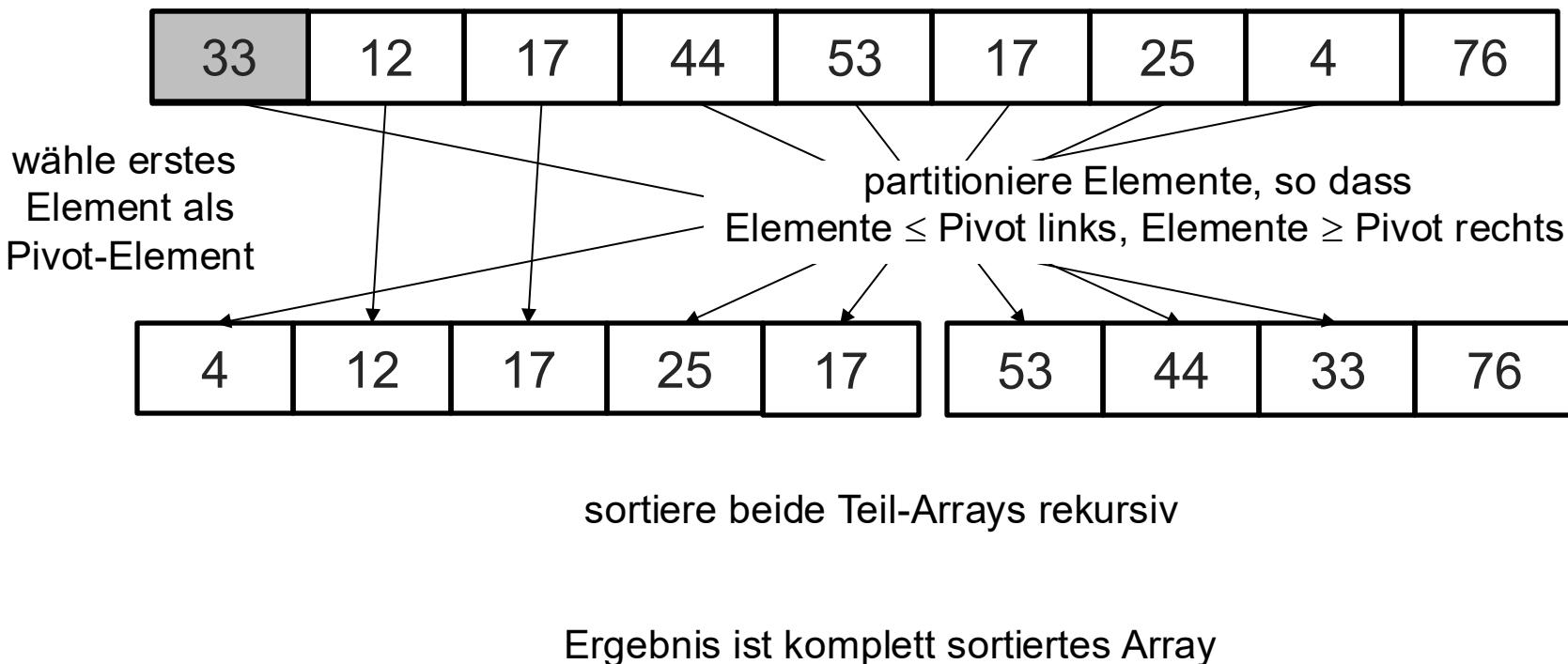
Lösen Sie folgende Rekursionsgleichung  
$$T(n) = 4 T(n/4) + n^2 \log n$$
  
mit Hilfe des Mastertheorems.

# Quicksort

# Idee

wie in Merge Sort verwendet Quicksort Divide & Conquer-Ansatz

Quicksort steckt mehr Arbeit in Aufteilen, Zusammenfügen kostenlos



# Algorithmus: Quicksort

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

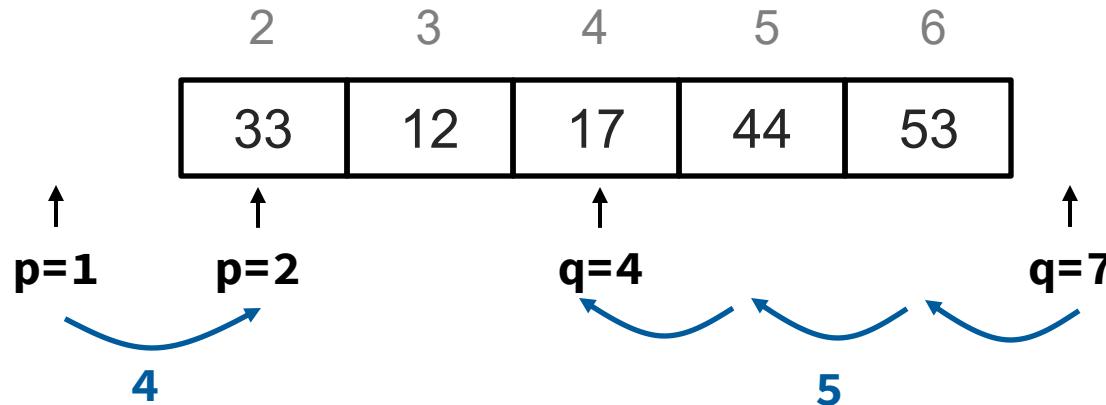
1 IF left<right THEN //more than one element
2   q=partition(A, left, right); // q partition index
3   quicksort(A, left, q);      // sort left part
4   quicksort(A, q+1, right);  // sort right part
```

```
partition(A, left, right) //req. left<right, ret. left..right-1

1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4   REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5   REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6   IF p<q THEN swap(A[p],A[q]);
7 return q                // A[left..q], A[q+1..right]
```

# Beispiel #1: Partition (I)

pivot=33  
left=2, right=6

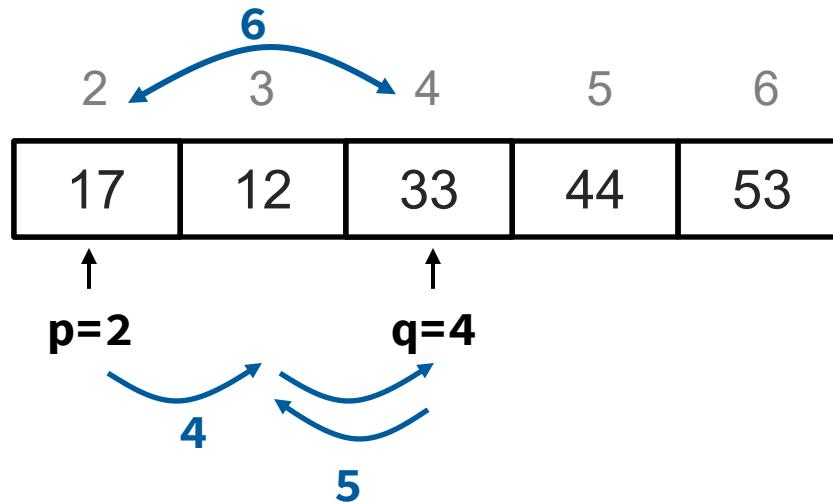


```
partition(A, left, right) //req.left<right, ret.left..right-1

1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7 return q                  // A[left..q], A[q+1..right]
```

# Beispiel #1: Partition (II)

**pivot=33**  
**left=2, right=6**

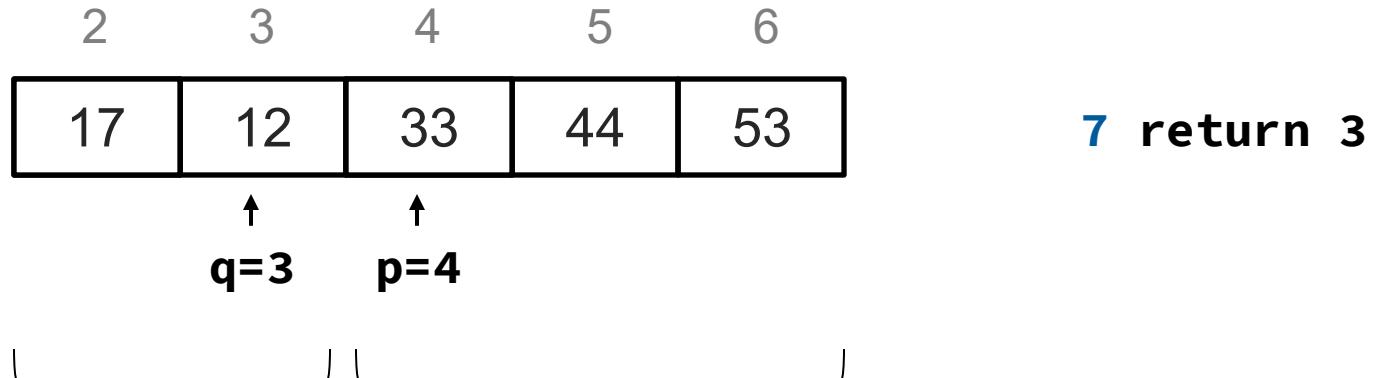


```
partition(A, left, right) //req.left<right, ret.left..right-1
```

```
1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7 return q                  // A[left..q], A[q+1..right]
```

# Beispiel #1: Partition (III)

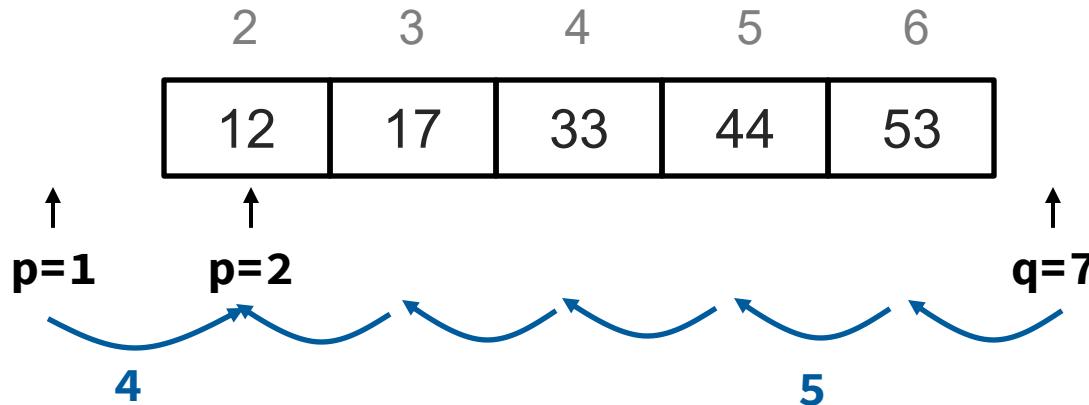
**pivot=33**  
**left=2, right=6**



```
partition(A, left, right) //req.left<right, ret.left..right-1  
  
1 pivot=A[left];  
2 p=left-1; q=right+1; //move from left resp. right  
3 WHILE p<q DO  
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up  
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down  
6     IF p<q THEN swap(A[p],A[q]);  
7 return q           // A[left..q], A[q+1..right]
```

# Beispiel #2: Partition (I)

pivot=12  
left=2, right=6

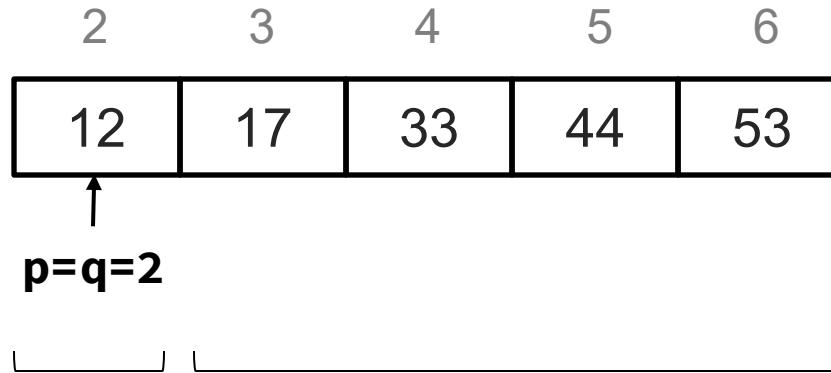


```
partition(A, left, right) //req.left<right, ret.left..right-1
```

```
1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7 return q                  // A[left..q], A[q+1..right]
```

# Beispiel #2: Partition (II)

**pivot=12**  
**left=2, right=6**

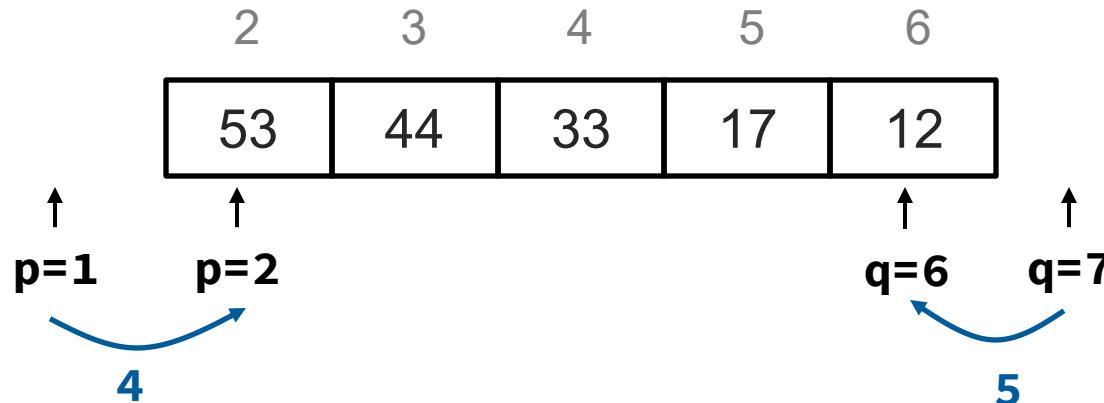


```
partition(A, left, right) //req.left<right, ret.left..right-1
```

```
1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7 return q                  // A[left..q], A[q+1..right]
```

# Beispiel #3: Partition (I)

**pivot=53**  
**left=2, right=6**

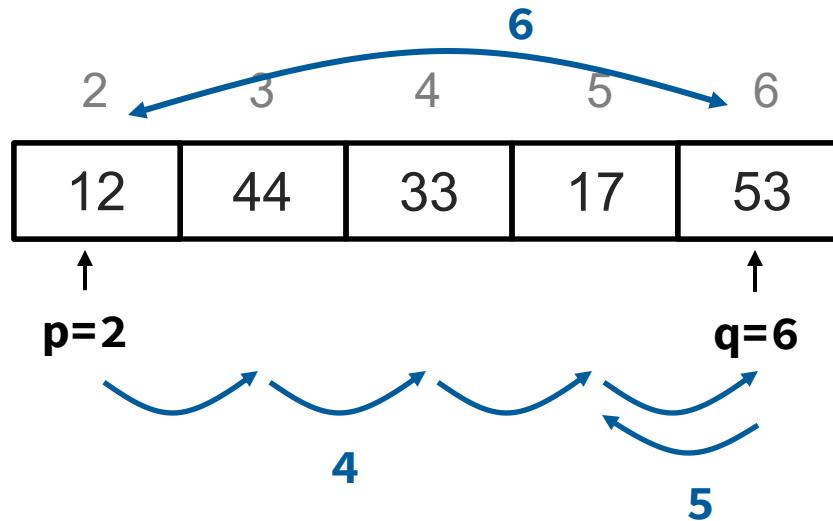


```
partition(A, left, right) //req.left<right, ret.left..right-1

1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7 return q                  // A[left..q], A[q+1..right]
```

# Beispiel #3: Partition (II)

**pivot=53**  
**left=2, right=6**

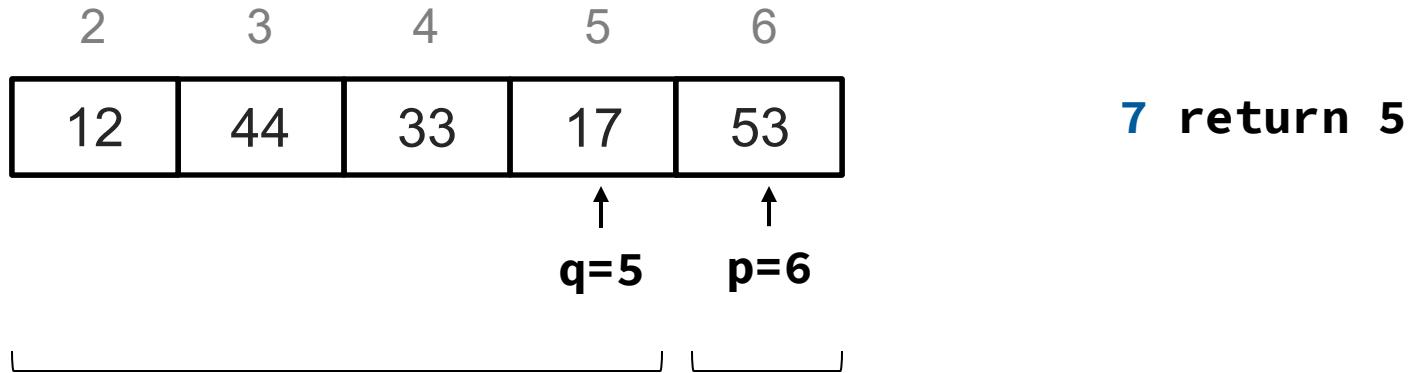


```
partition(A, left, right) //req.left<right, ret.left..right-1
```

```
1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7 return q                  // A[left..q], A[q+1..right]
```

# Beispiel #3: Partition (III)

**pivot=53**  
**left=2, right=6**



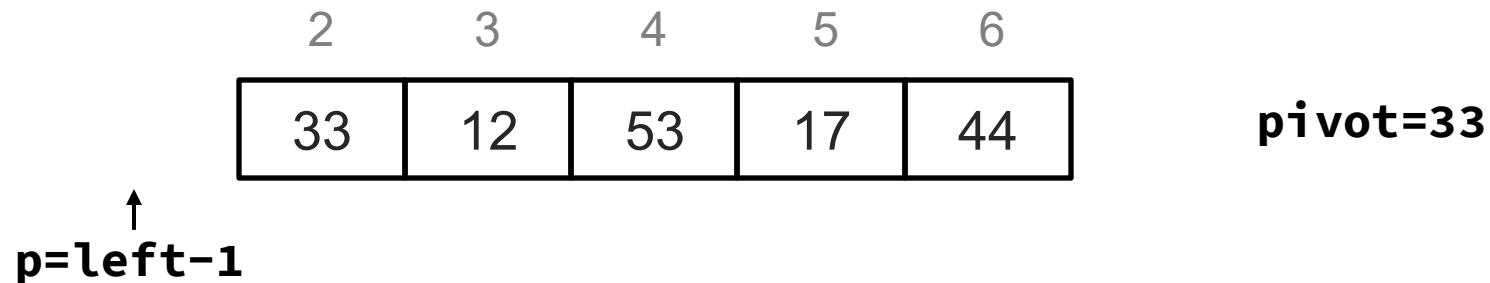
```
partition(A, left, right) //req.left<right, ret.left..right-1  
  
1 pivot=A[left];  
2 p=left-1; q=right+1; //move from left resp. right  
3 WHILE p<q DO  
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up  
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down  
6     IF p<q THEN swap(A[p],A[q]);  
7 return q           // A[left..q], A[q+1..right]
```

# Partition Terminierung (I)

Betrachte **REPEAT**-Schleife 4

Behauptung: Vor Eintritt in 4 steht rechts von **p** stets Element  $\geq \text{pivot}$

Gilt zu Beginn (erste Ausführung **WHILE**-Schleife),  
da **p=left-1** und **A[left]=A[p+1]=pivot**



# Partition Terminierung (II)

Betrachte **REPEAT**-Schleife 4

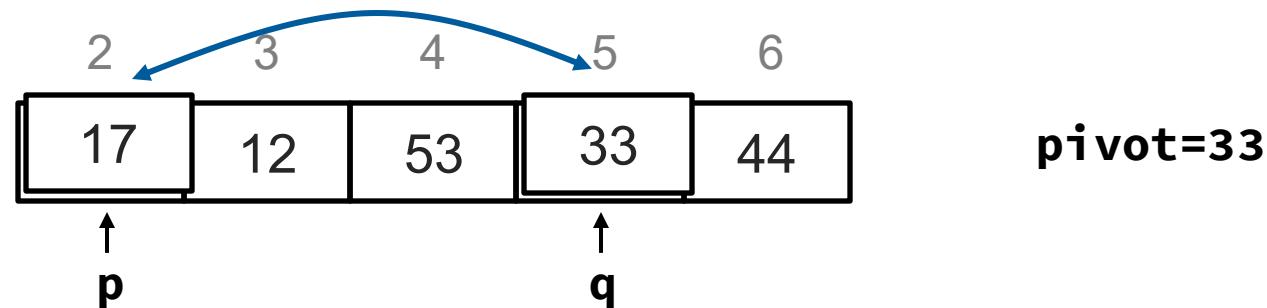
Behauptung: Vor Eintritt in 4 steht rechts von **p** stets Element  $\geq \text{pivot}$

Annahme: **WHILE**-Schleife bereits mindestens einmal ausgeführt

Dann zeigt **p** nach **REPEAT** in voriger **WHILE**-Iteration auf Wert  $\geq \text{pivot}$

In aktueller Iteration wird **REPEAT** nur erneut erreicht, wenn  $\text{p} < \text{q}$  und somit in voriger **WHILE**-Iteration **swap** ausgeführt wurde

**swap** tauschte  $\mathbf{A[p]} >= \text{pivot}$  weiter nach rechts ( $\text{p} < \text{q}$ ), daraus folgt Behauptung



# Partition Terminierung (III)

Analog: Vor Eintritt in **5** steht links von **q** stets ein Element  $\leq \text{pivot}$

Folglich terminieren beide **REPEATs** in jeder Iteration der **WHILE**-Schleife

Da in jeder Iteration der **WHILE**-Schleife **p** (bzw. **q**) um mindestens 1 erhöht (bzw. erniedrigt) wird, muss irgendwann **p}={q** sein und die **WHILE**-Schleife terminieren

```
partition(A,left,right) //req.left<right,ret.left..right-1

1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p],A[q]);
7 return q                // A[left..q], A[q+1..right]
```

# Partition Korrektheit (I)

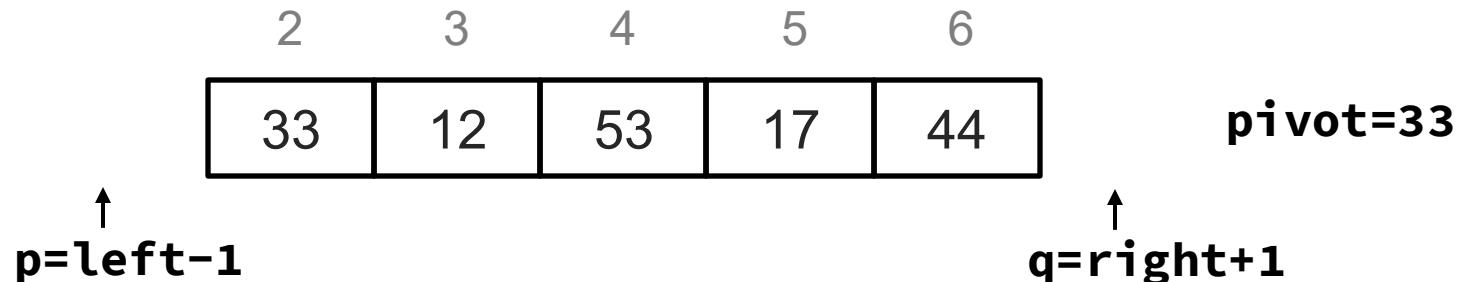
zz.  $A[\text{left} \dots q] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$   
 $q$  Rückgabewert

Schleifeninvariante: Bei Eintritt in Zeile 4 der **WHILE**-Schleife enthalten  
 $A[\text{left} \dots p]$  nur Elemente  $\leq \text{pivot}$  und  $A[q \dots \text{right}]$  nur  $\geq \text{pivot}$

vor den **REPEATs**

Induktionsbasis:

Gilt vor erstem Eintritt für „leere“ Arrays mit  $p=\text{left}-1$  und  $q=\text{right}+1$



# Partition Korrektheit (II)

zz.  $A[\text{left} \dots q] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$   
 $q$  Rückgabewert

Schleifeninvariante: Bei Eintritt in Zeile 4 der **WHILE**-Schleife enthalten  
 $A[\text{left} \dots p]$  nur Elemente  $\leq \text{pivot}$  und  $A[q \dots \text{right}]$  nur  $\geq \text{pivot}$

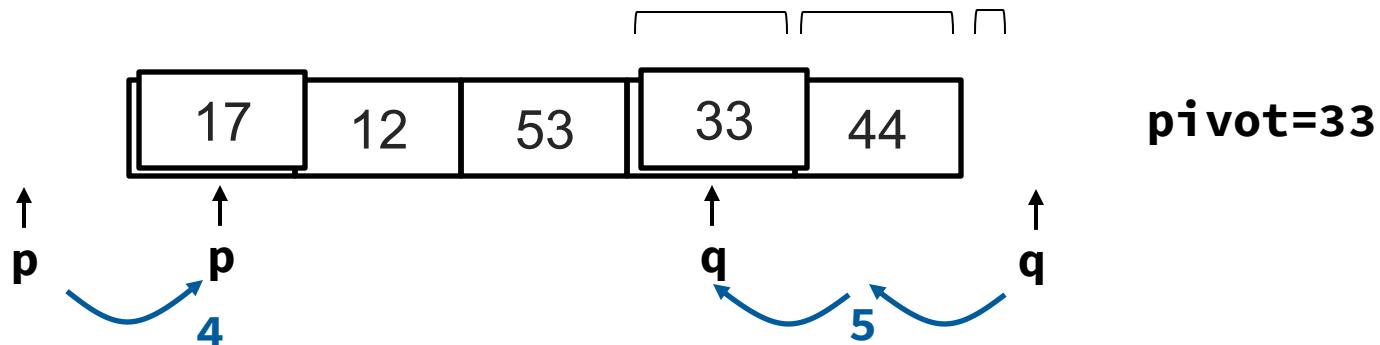
**Induktionsschritt:** Wenn wieder in Zeile 4, muss (noch) gelten  $p < q$

In voriger Iteration:

$p$  lief in 4 nur über Werte  $< \text{pivot}$ ,  $q$  in 5 nur über Werte  $> \text{pivot}$ ,  
bis schließlich  $A[p] \geq \text{pivot}$  und  $A[q] \leq \text{pivot}$

Folgender **swap** wegen  $p < q$  tauschte beide Werte, Behauptung folgt

swap:  $\geq \text{pivot}$      $> \text{pivot}$     Ind.vor.:  $\geq \text{pivot}$



# Partition Korrektheit (III)

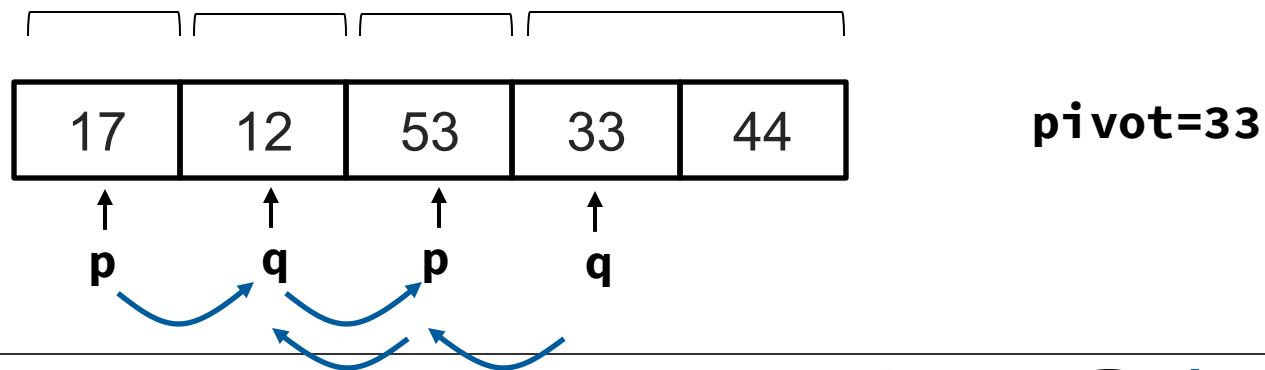
zz.  $A[\text{left} \dots q] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$   
 $q$  Rückgabewert

Betrachte letzte Iteration von **WHILE**, in der  $q \leq p$  wird

Bei Eintritt  $A[\text{left} \dots p] \leq \text{pivot}$ ,  $A[q \dots \text{right}] \geq \text{pivot}$  wegen Invariante,  
 $p$  läuft in 4 nur über Werte  $< \text{pivot}$ ,  $q$  in 5 nur über Werte  $> \text{pivot}$ ,  
bis schließlich  $A[p] \geq \text{pivot}$  und  $A[q] \leq \text{pivot}$

Dann  $A[\text{left} \dots p-1] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$  nach **REPEATs**

Invariante:  $\leq \text{pivot}$        $< \text{pivot}$        $> \text{pivot}$       Invariante:  $\geq \text{pivot}$



# Partition Korrektheit (III)

zz.  $A[\text{left} \dots q] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$   
 $q$  Rückgabewert

Betrachte letzte Iteration von **WHILE**, in der  $q \leq p$  wird

Dann  $A[\text{left} \dots p-1] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$  nach **REPEATs**

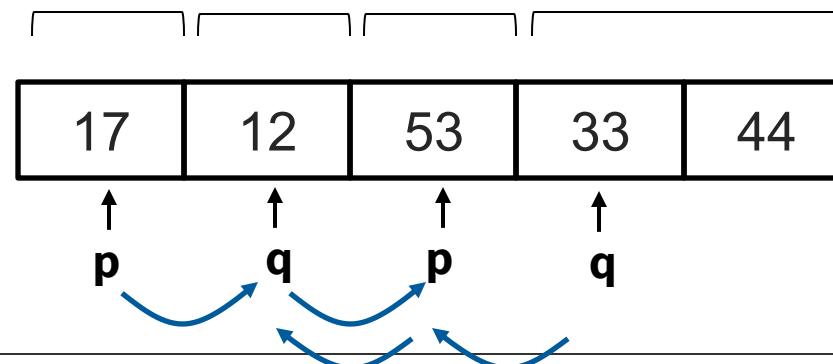
Da Abbruch von **REPEAT**, wenn  $A[p] > \text{pivot}$  bzw. wenn  $A[q] \leq \text{pivot}$ , entweder  $q=p$  (Beispiel #2) oder  $p=q+1$  (Beispiel #3/hier)

Im Fall  $p=q+1$  gilt also  $A[\text{left} \dots q] = A[\text{left} \dots p-1] \leq \text{pivot}$  und  $A[p \dots \text{right}] = A[q+1 \dots \text{right}] \geq \text{pivot}$

Invariante:  $\leq \text{pivot}$

$< \text{pivot}$

Invariante:  $\geq \text{pivot}$



**pivot=33**

**7 return q**

# Partition Korrektheit (IV)

zz.  $A[\text{left} \dots q] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$   
 $q$  Rückgabewert

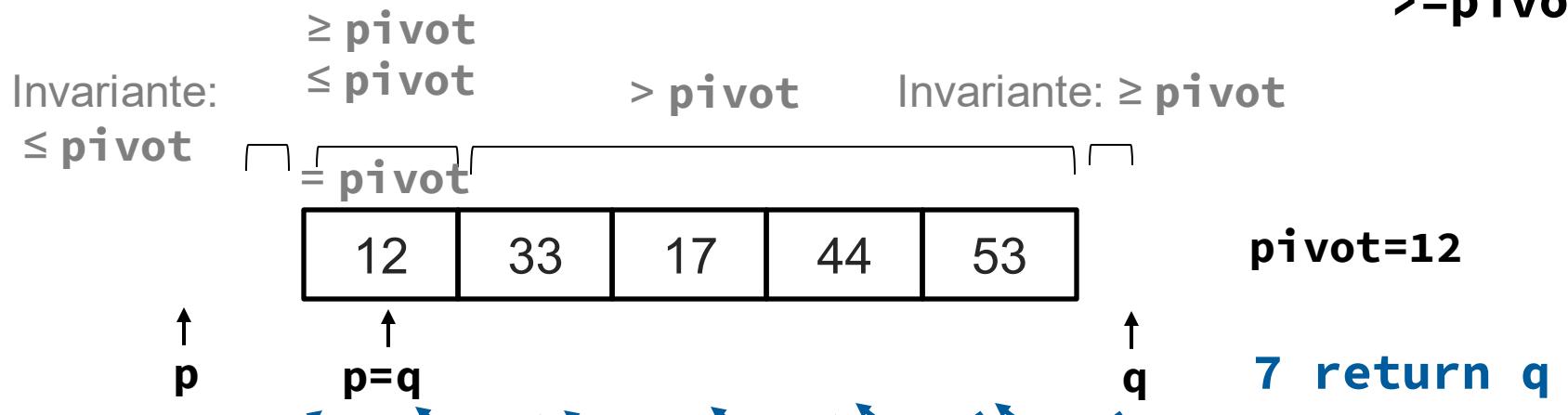
Betrachte letzte Iteration von **WHILE**, in der  $q \leq p$  wird

Dann  $A[\text{left} \dots p-1] \leq \text{pivot}$ ,  $A[q+1 \dots \text{right}] \geq \text{pivot}$  nach **REPEATs**

Da Abbruch von **REPEAT**, wenn  $A[p] >= \text{pivot}$  bzw. wenn  $A[q] \leq \text{pivot}$ , entweder  $q=p$  (Beispiel #2/hier) oder  $p=q+1$  (voriges Beispiel)

Im Fall  $q=p$  gilt  $A[p]=A[q]=\text{pivot}$  und daher

$A[\text{left} \dots p] = A[\text{left} \dots q] \leq \text{pivot}$ ,  $A[p+1 \dots \text{right}] = A[q+1 \dots \text{right}] \geq \text{pivot}$



# Partition Korrektheit (V)

noch zz.:  $\text{left} \leq q < \text{right}$ ,  
nicht-triviale Aufteilung

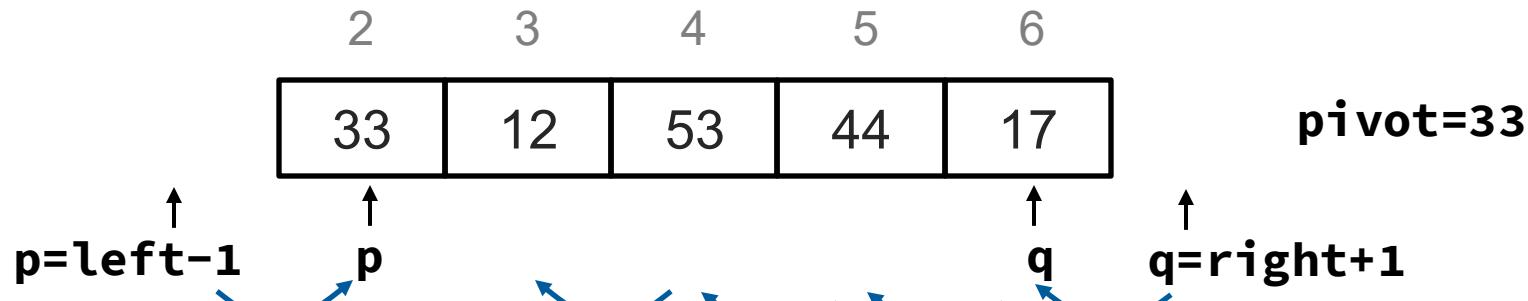
$p=q=\text{right}$  kann nur passieren, wenn **WHILE**-Schleife nur einmal ausgeführt; sonst würde nächste Iteration von **WHILE** Wert  $q$  weiter erniedrigen

Andererseits  $p=\text{left}$  nach 1. Iteration von **WHILE**, da  $A[\text{left}] = \text{pivot}$

Wegen  $p=\text{left} < \text{right}$  würde  $q=\text{right}$  weitere **WHILE**-Iteration bedeuten

nur solche Eingaben  
in **Partition** erlaubt

Also definitiv  $q < \text{right}$

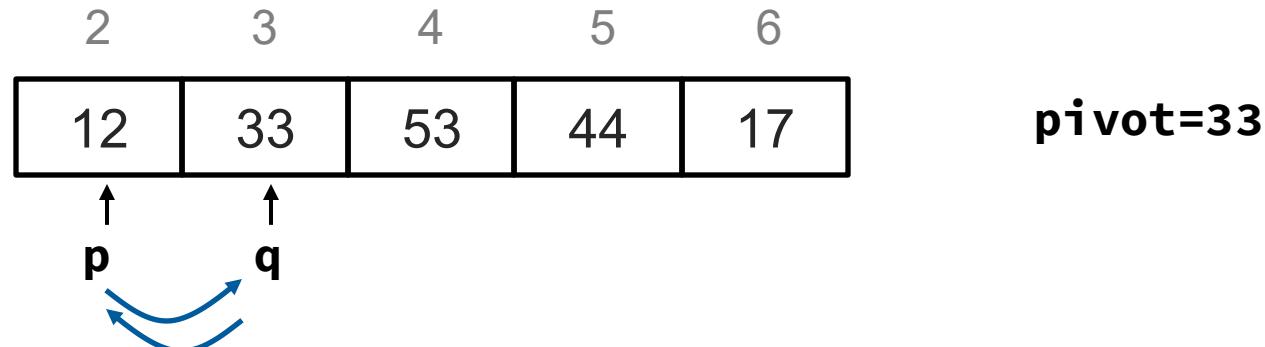


# Partition Korrektheit (VI)

noch zz.:  $\text{left} \leq \text{q} < \text{right}$ ,  
nicht-triviale Aufteilung

Zusätzlich kann  $\text{q} < \text{left}$  nicht passieren, da

in allen **WHILE**-Iterationen (auch in letzter, in der  $\text{q} \leq \text{p}$  wird)  
vor **REPEAT** links von  $\text{q}$  immer Wert  $\leq \text{pivot}$  steht (vgl. Terminierung),  
**REPEAT** also nur bis  $\text{left}$  runterzählen kann



# Partition Laufzeit

Für jede Erhöhung/Erniedrigung von **p** bzw. **q** konstant viele Schritte

**p** und **q** haben zu Beginn Abstand  $n + 2$   
und bewegen sich in jeder Iteration aufeinander zu  $O(n)$

**p** und **q** bewegen sich in jeder einzelnen  
**REPEAT**-Iteration maximal 1 aufeinander zu  $\Omega(n)$

```
partition(A, left, right) //req.left<right,  
1 pivot=A[left];  
2 p=left-1; q=right+1; //move from left resp. right  
3 WHILE p<q DO  
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up  
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down  
6     IF p<q THEN Swap(A[p],A[q]);  
7 return q           // A[left..q], A[q+1..right]
```

Laufzeit Partition  
 $\Theta(n)$

# **Laufzeitanalyse Quicksort: Worst-Case, Average-Case und erwartete Laufzeit**

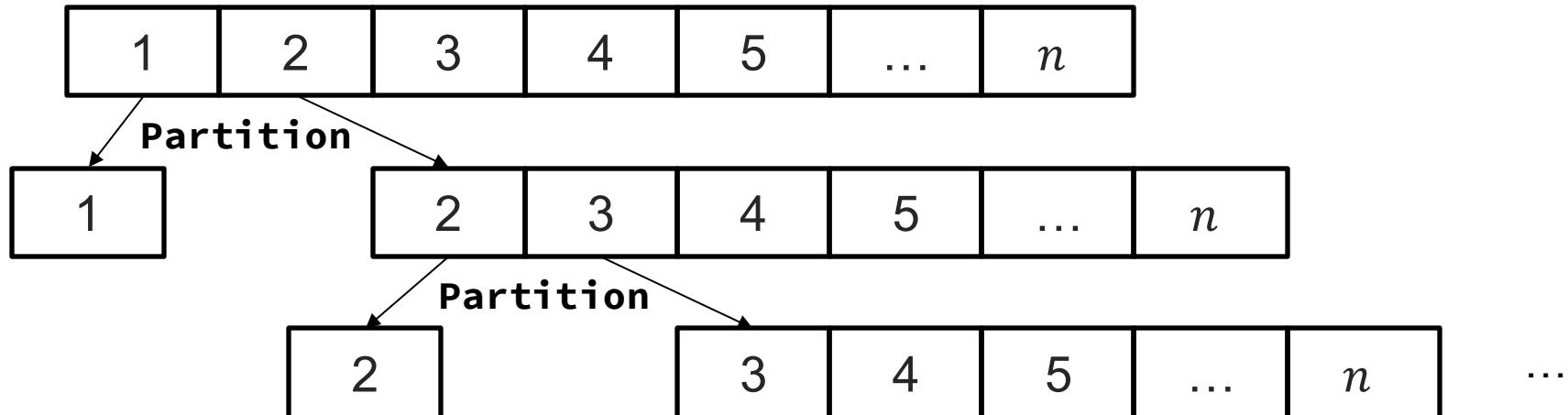
# Laufzeit im Worst-Case: Untere Schranke

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

```
1 IF left < right THEN //more than one el
2   q=partition(A, left, right);    // q pa
3   quicksort(A, left, q);        // sort
4   quicksort(A, q+1, right);    // sort right part
```

( $n - 1$ )-mal **Partition**  
ergibt Quicksort  
Gesamtlaufzeit  $\Omega(n^2)$

Ungünstiges Array: **Partition** spaltet immer nur ein Element ab (Bsp #2)



# Laufzeit im Worst-Case: Obere Schranke (I)

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

```
1 IF left<right THEN //more than one element
2   q=partition(A, left, right); // q partition index
3   quicksort(A, left, q);      // sort left part
4   quicksort(A, q+1, right);  // sort right part
```

Intuition: nur ein Element abzuspalten ist auch schlechtester Fall und daher

$$\begin{aligned} T(n) &\leq T(n-1) + dn \quad \text{---} \quad \text{Aufwand für } T(1), \textbf{IF} \\ &\leq T(n-2) + d(n-1) + dn \\ &\quad \vdots \\ &\leq \sum_{i=1}^n di = \mathcal{O}(n^2) \end{aligned}$$

# Laufzeit im Worst-Case: Obere Schranke (II)

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

```
1 IF left < right THEN //more than one el
2   q=partition(A, left, right); // q pa
3   quicksort(A, left, q); // sort
4   quicksort(A, q+1, right); // sort right part
```

Quicksort  
(Worst-Case-)Laufzeit  
 $\Theta(n^2)$

Formal per Induktion: Behauptung  $T(n) \leq dn^2$

**Basisfall:** gilt für  $n = 1$ , da  $T(1) \leq dn$

$i$  nicht-trivialer  
Partitionsindex (daher  
Induktion anwendbar)

**Induktions-  
schritt:**

$$\begin{aligned} T(n) &\leq \max_{i=1,\dots,n-1} (T(n-i) + T(i)) + dn \\ &\leq \max_{i=1,\dots,n-1} (d(n-i)^2 + di^2) + dn \\ &\leq d(n-1)^2 + d + dn \\ &\leq dn^2 - 2dn + d + d + dn \\ &\leq dn^2 \quad \text{für } n \geq 2 \end{aligned}$$

maximal für  $i = 1$

# Best-Case für Quicksort (I)

Im besten Fall Aufteilung in gleichgroße Arrays wie bei Merge Sort:

$$T(n) = 2T(n/2) + \Theta(n) \Rightarrow \text{Laufzeit } \Theta(n \log n)$$

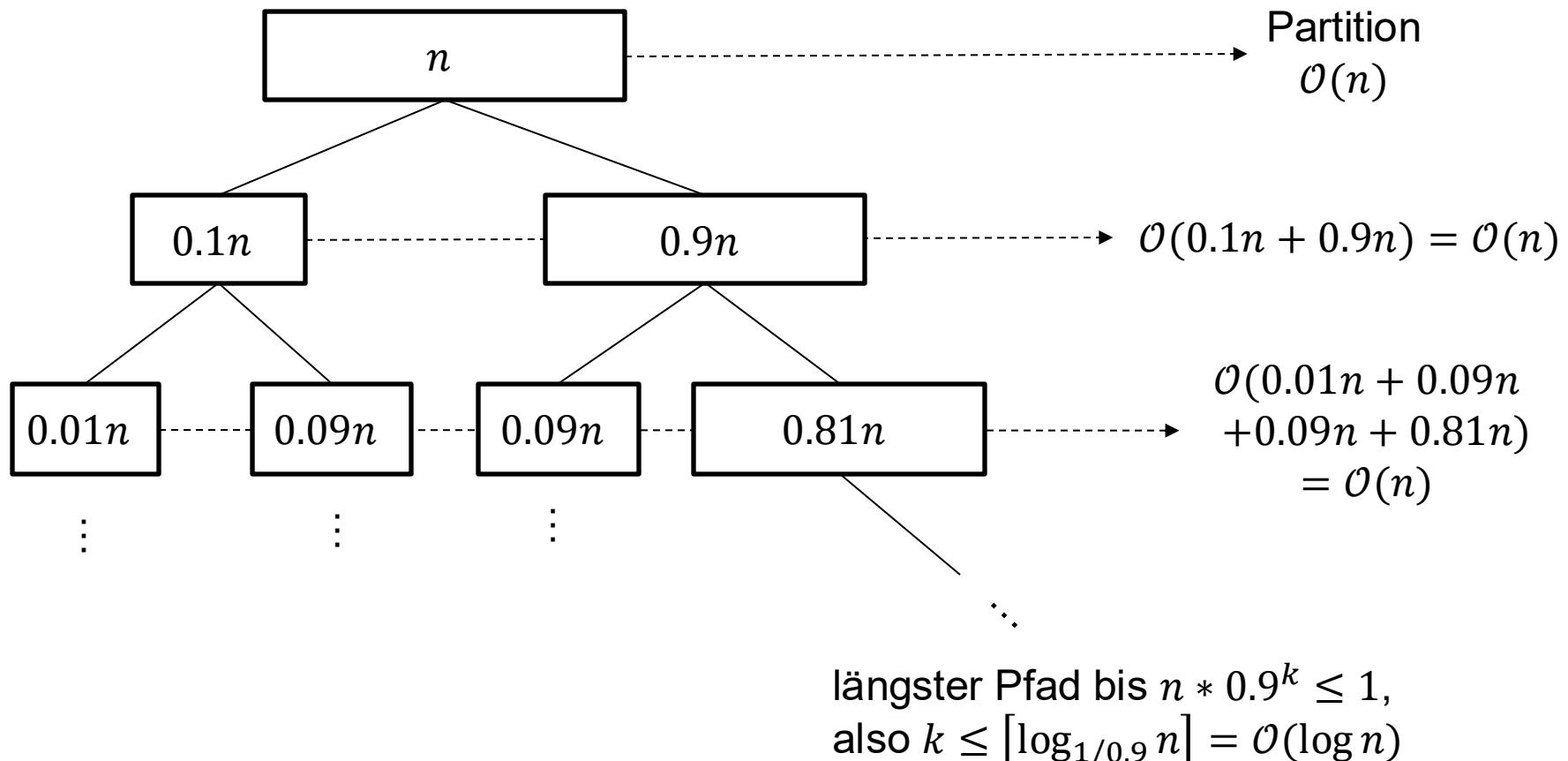
Gilt auch, solange beide Arrays in Größenordnung  $\Omega(n)$ , z.B. stets 10% der  $n$  Elemente links und 90% rechts:

$$T(n) = T(0.1n) + T(0.9n) + \Theta(n) \Rightarrow \text{Laufzeit } \Theta(n \log n)$$

# Best-Case für Quicksort (II)

„Rekursionsbaum“

Anzahl zu sortierender Elemente



auf jeder Ebene Aufwand  $\mathcal{O}(n) \Rightarrow$  Gesamtaufwand  $\mathcal{O}(n \log n)$

# Average-Case-Laufzeiten?

(Worst-Case-)Laufzeit  
 $T(n) = \max \{ \# \text{Schritte f\"ur } x \}$

Achtung: \"ubliche  
Definition ist komplizierter

Intuitiver Ansatz:  
 $T(n) = E_{D(n)} [ \# \text{Schritte f\"ur } x ]$

erwartete Anzahl von Schritten \"uber  
Verteilung  $D(n)$  auf Eingabedaten  
der Komplexit\"at  $n$

Wie verh\"alt sich Quicksort im Durchschnitt auf „zuf\"alliger“ Eingabe?

F\"ur zuf\"allige Permutation  $D(n)$  eines fixen Arrays von  $n$  Elementen  
ben\"otigt Quicksort  $E_{D(n)} [ \text{Anzahl Schritte} ] = \mathcal{O}(n \log n)$

Aber was ist eine realistische Verteilung auf Eingaben???

# Randomisierte Variante Quicksort

Ansatz: betrachte statt zufälliger Eingabe randomisierte Variante  
**randomizedQuicksort**

wählt als Pivot-Element immer uniform eines der Elemente  
(und tauscht es an Anfang des Arrays, um wie zuvor fortzufahren)

```
partition(A, left, right) //req.left<right, ret.left..right-1
0 j=RANDOM(left, right); swap(A[left], A[j]);
                           //j uniform in [left..right]
1 pivot=A[left];
2 p=left-1; q=right+1;   //move from left resp. right
3 WHILE p<q DO
4     REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5     REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6     IF p<q THEN swap(A[p], A[q]);
7 return q                  // A[left..q], A[q+1..right]
```

# Erwartete Laufzeit

Achtung: manchmal auch als Average-Case-Laufzeit bezeichnet

(Worst-Case-)Laufzeit

$$T(n) = \max \{ \# \text{Schritte f\"ur } x \}$$

Erwartete Laufzeit

$$T(n) = \max \{ E_A [ \# \text{Schritte f\"ur } x ] \}$$

erwartete Anzahl von Schritten  
(\"uber zuf\"allige Wahl des Algorithmus A)  
f\"ur „schlechteste“ Eingabe  
der Komplexit\"at  $n$

Erwartete Laufzeit von Randomized-Quicksort ist  $\mathcal{O}(n \log n)$

Intuition:

zuf\"allige Wahl des Pivot-Elementes teilt Array im Durchschnitt mittig,  
unabh\"angig davon, wie Array aussieht

# Merge Sort vs. Randomized-Quicksort (I)

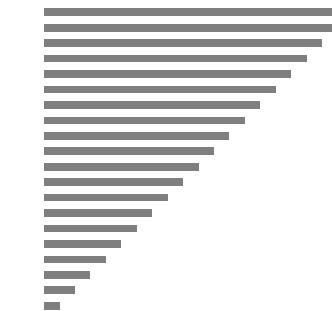
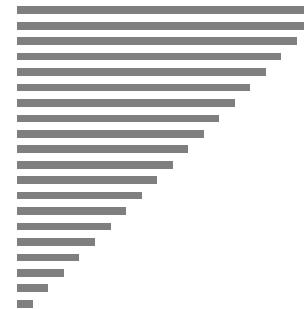
„unsortierte  
Eingabe“

Merge Sort



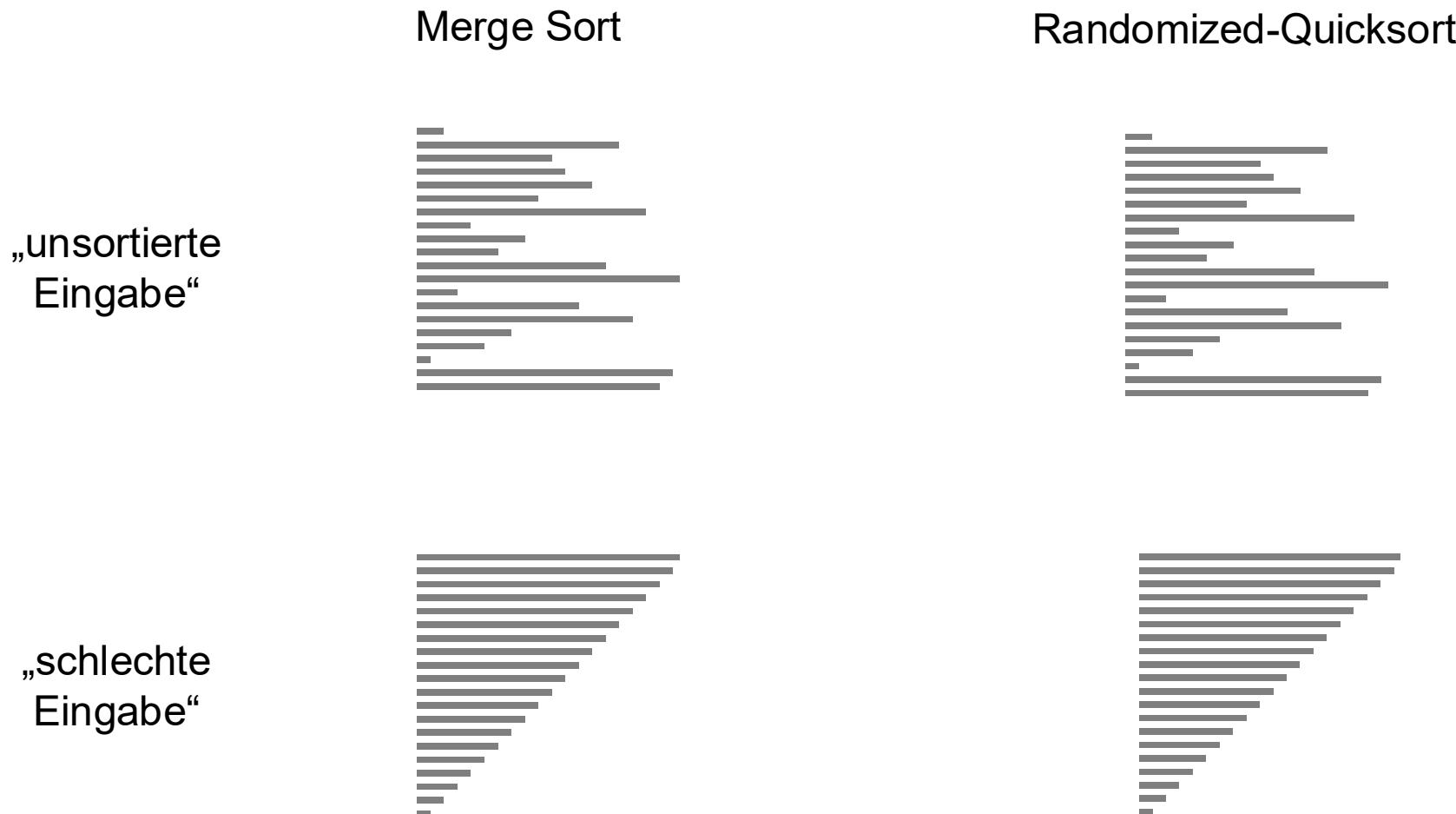
„schlechte  
Eingabe“

Randomized-Quicksort



Quelle: <https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/>

# Merge Sort vs. Randomized-Quicksort (II)



Quelle: <https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/>

# Vergleich: Insertion, Merge, und Quicksort

Insertion Sort	Merge Sort	Quicksort
Laufzeit $\Theta(n^2)$	Beste asymptotische Laufzeit $\Theta(n \log n)$	Worst-Case-Laufzeit $\Theta(n^2)$ , randomisiert mit erwarteter Laufzeit $\Theta(n \log n)$
einfache Implementierung		in Praxis meist schneller als Merge Sort, da weniger Kopieroperationen
für kleine $n \leq 50$ oft beste Wahl		Implementierungen schalten für kleine $n$ meist auf Insertion Sort

# Sortieren in Java

## sort

```
public static void sort(byte[] a)
```

Sorts the specified array into ascending numerical order.

### Implementation Note:

The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers  $O(n \log(n))$  performance on all data sets, and is typically faster than traditional (one-pivot) Quicksort implementations.

### Parameters:

a - the array to be sorted

Quelle: [docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/Arrays.html](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/Arrays.html)

Dual-Pivot-Quicksort: Drittelt Array gemäß zweier Pivot-Elemente



Ist **randomizedQuicksort** im engeren Sinne  
überhaupt ein Algorithmus?



Wieviel Schritte braucht **randomizedQuicksort**  
im schlechtesten Fall?

# **Untere Schranke für vergleichsbasiertes Sortieren**

(hier nur für deterministische Algorithmen,  
gilt aber auch für randomisierte Algorithmen im Durchschnitt)

# Vergleichsbasiertes Sortieren

```
sortByComp(n)
// returns array I with sorted indexes:
// A[ I[i] ] <= A[ I[i+1] ] for i=0,...,n-1
```

```
1 done=false;
2 WHILE !done DO
3     determine (i,j); // arbitrarily
4     comp(i,j); // returns A[i] <= A[j]?
5     set done; // true or false
6 compute I from comp-information only;
7 return I
```

kennt nur Größe n  
des Eingabe-Arrays A

Informationen über A nur  
durch Vergleichsresultate  
(Ja/Nein) für gewählte  
Indizes i, j

Alle Sortieralgorithmen bisher sind vergleichsbasiert

# Untere Schranke

```
sortByComp(n)
// returns array I with sorted indexes:
// A[ I[i] ] <= A[ I[i+1] ] for i=0,...,n-1

1 done=false;
2 WHILE !done DO
3     determine (i,j); // arbitrarily
4     comp(i,j); // returns A[i] <= A[j]?
5     set done; // true or false
6     compute I from comp-information only;
7 return I
```

Theorem: Jeder (korrekte) vergleichsbasierte Sortieralgorithmus muss mindestens  $\Omega(n \cdot \log n)$  viele Vergleiche machen

# Untere Schranke: Eingabemenge

Betrachte Ausgangs-Array  $A$  mit  $A[i] = i$

0	1	...					n-1
---	---	-----	--	--	--	--	-----

und jede Permutation davon,  $\pi(A)$ , für alle  $\pi$ :

$\pi(0)$	$\pi(1)$	...					$\pi(n-1)$
----------	----------	-----	--	--	--	--	------------

Insgesamt also  $n!$  viele mögliche Eingabe-Arrays

Jedes Eingabe-Array erfordert andere Ausgabe  $I = \pi^{-1}$

# Mögliche Ausgaben

```
sortByComp(n)
```

```
// returns array I with sorted indexes:  
// A[ I[i] ] <= A[ I[i+1] ] for i=0,...,n-1
```

```
1 done=false;  
2 WHILE !done DO  
3     determine (i,j); // arbitrarily  
4     comp(i,j); // returns A[i] <= A[j]?  
5     set done; // true or false  
6 compute I from comp-information only;  
7 return I
```

Annahme:  
macht  $m$  Vergleiche

es gibt maximal  
 $2^m$  mögliche  
Antwortsequenzen  
Ja/Nein

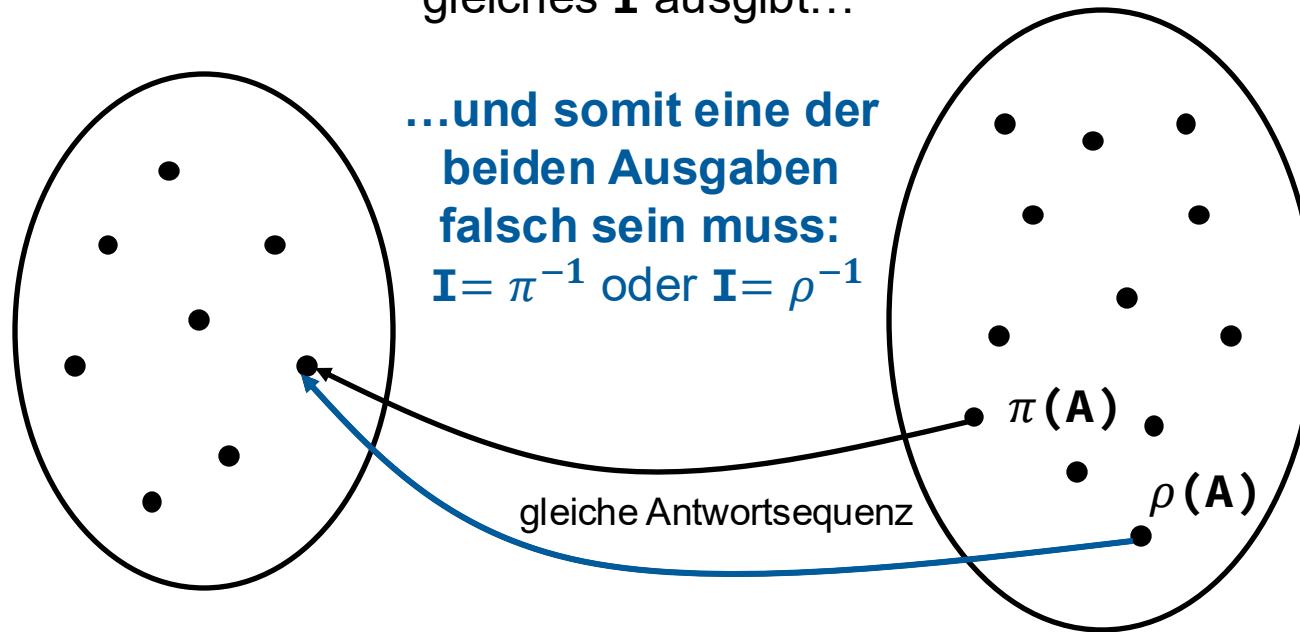
einzige Info  
über A  
(außer n)

Algorithmus gibt  
maximal  $2^m$  verschiedene  
Ausgaben I

Determiniertheit:  
gleiche Antwortsequenzen  
(selbst für verschiedene Arrays)  
führen zu gleicher Ausgabe

# Ein- und Ausgabeverhältnis

Wenn  $2^m < n!$ , dann gibt es verschiedene Arrays  $\pi(\mathbf{A})$  und  $\rho(\mathbf{A})$  für die der Algorithmus gleiches  $\mathbf{I}$  ausgibt...



Algorithmus gibt maximal  $2^m$  verschiedene Ausgaben  $\mathbf{I}$

Es gibt  $n!$  verschiedene Eingabe-Arrays  $\pi(\mathbf{A})$

# Schranke ausrechnen

Wenn  $2^m < n!$ , dann gibt es verschiedene Arrays  $\pi(\mathbf{A})$  und  $\rho(\mathbf{A})$  für die der Algorithmus gleiches  $\mathbf{I}$  ausgibt...

Es muss also  $2^m \geq n!$  gelten bzw.  $m \geq \log(n!)$

$$\downarrow \quad \begin{aligned} &\text{Stirling-Approximation:} \\ &n! \geq \left(\frac{n}{e}\right)^n \end{aligned}$$

$$\text{bzw. } m = \Omega(n \cdot \log(n))$$

Theorem: Jeder (korrekte) vergleichsbasierte Sortieralgorithmus muss mindestens  $\Omega(n \cdot \log n)$  viele Vergleiche machen



Warum ist Quicksort mit seinen Vertauschungen vergleichsbasiert?

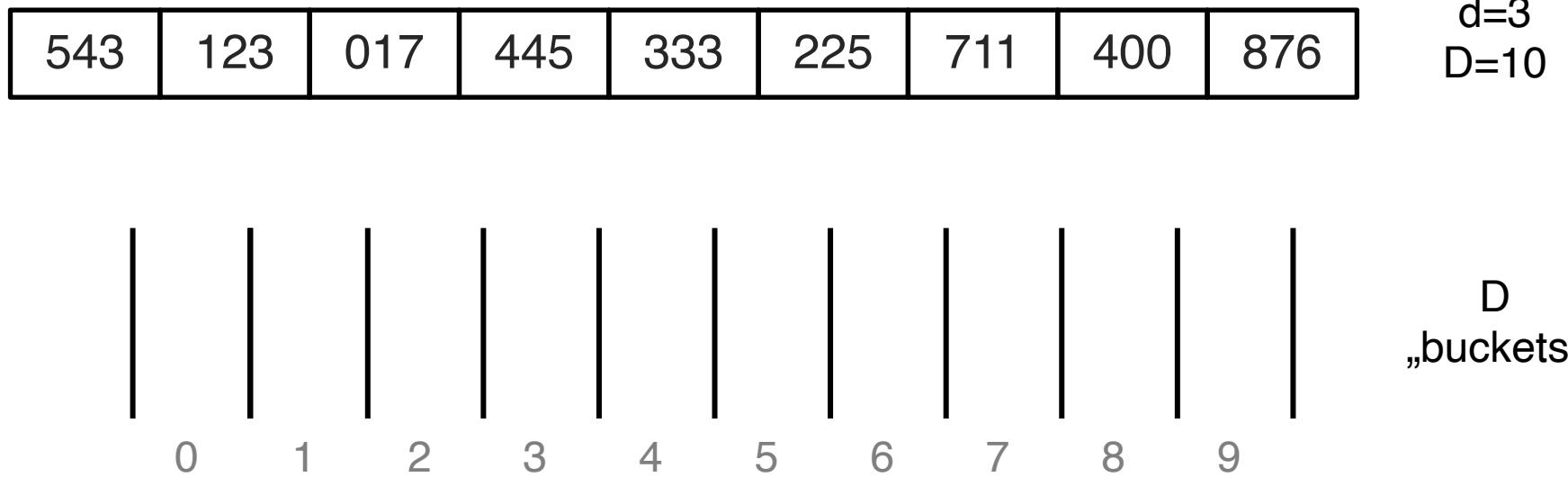


Können Sie sich eine Operation in einem Algorithmus vorstellen, die nicht kompatibel mit der Eigenschaft „vergleichsbasiert“ ist?

# **RadixSort:** **Sortieren in linearer Zeit (?)**

# Ansatz

Schlüssel sind d-stellige Werte in D-närem Zahlensystem:

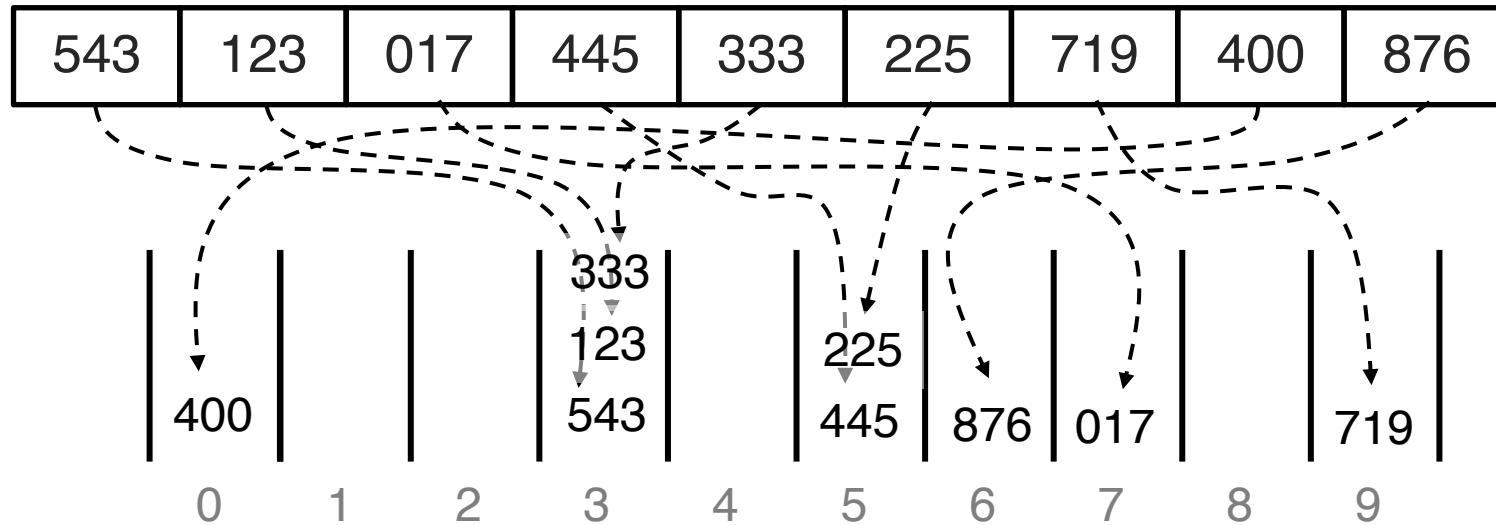


„Buckets“ erlauben  
Einfügen und dann Entnehmen (in eingefügter Reihenfolge)  
in konstanter Zeit

Queues  
(FIFO-Systeme)  
→ Abschnitt 3

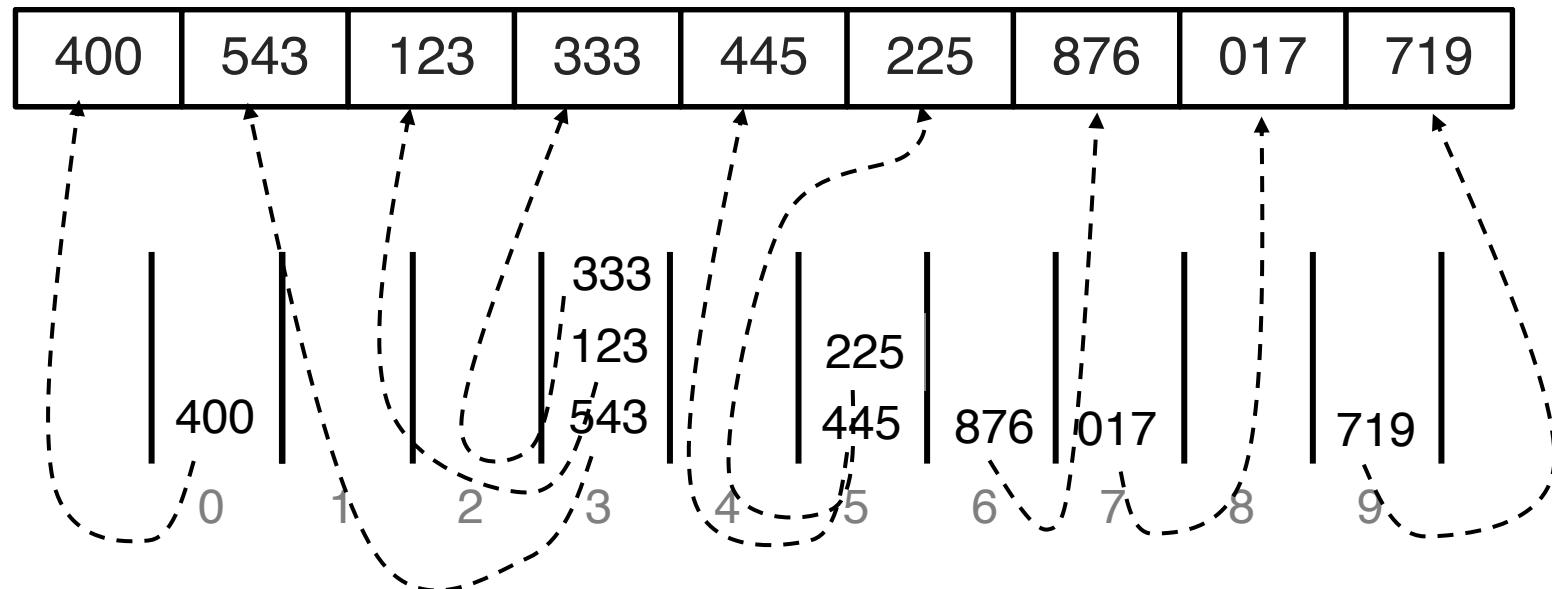
# Erste Iteration (I)

Gehe Array von links nach rechts durch und füge Werte entsprechend **kleinstwertigster** Ziffer in Bucket ein:



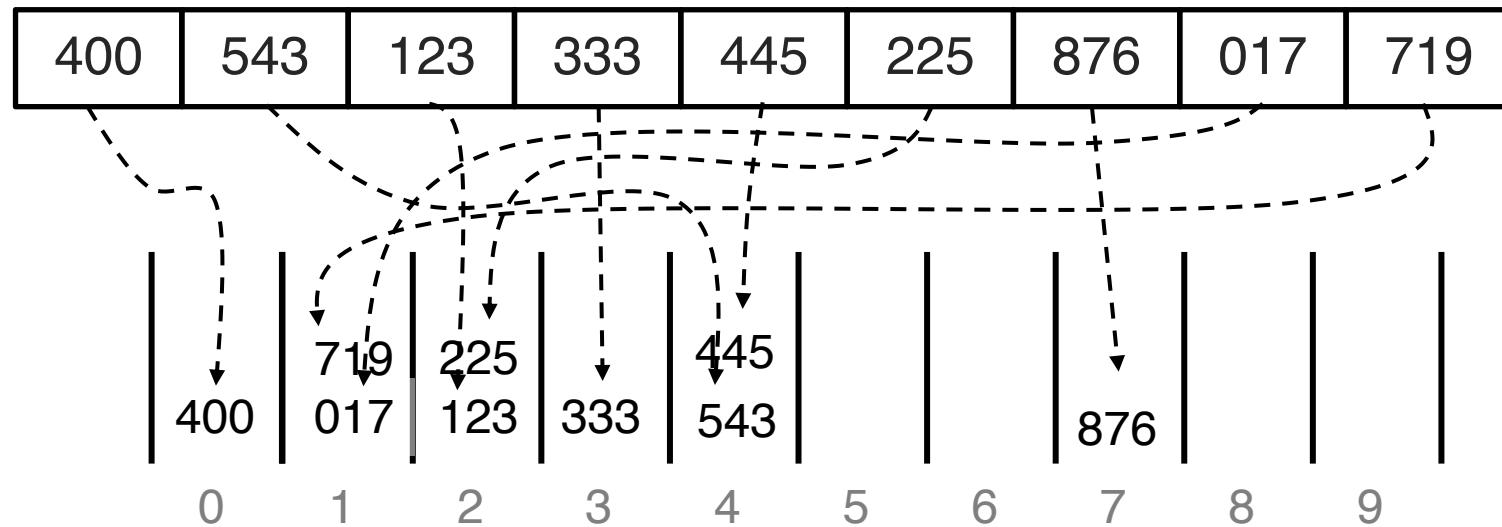
# Erste Iteration (II)

Gehe Buckets von links nach rechts durch, entnimm Werte und füge Werte an nächster Stelle im Array ein:



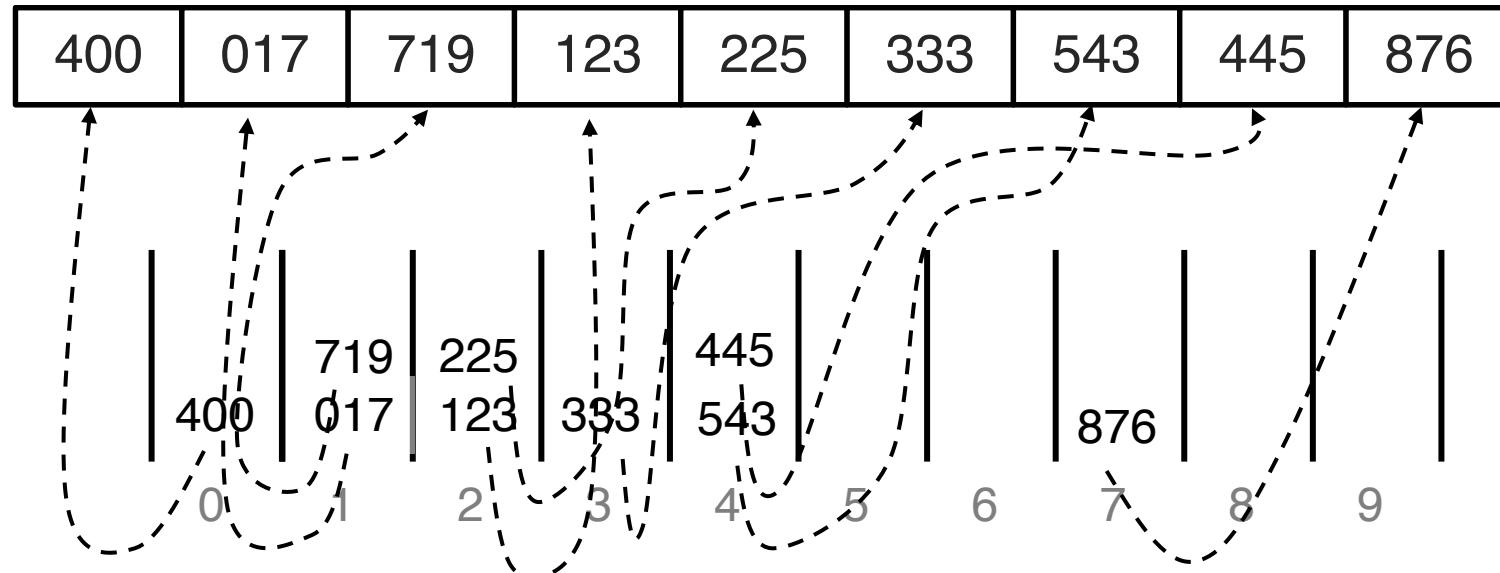
# Zweite Iteration (I)

Gehe Array von links nach rechts durch und füge Werte entsprechend **zweitkleinstwertigster** Ziffer in Bucket ein:



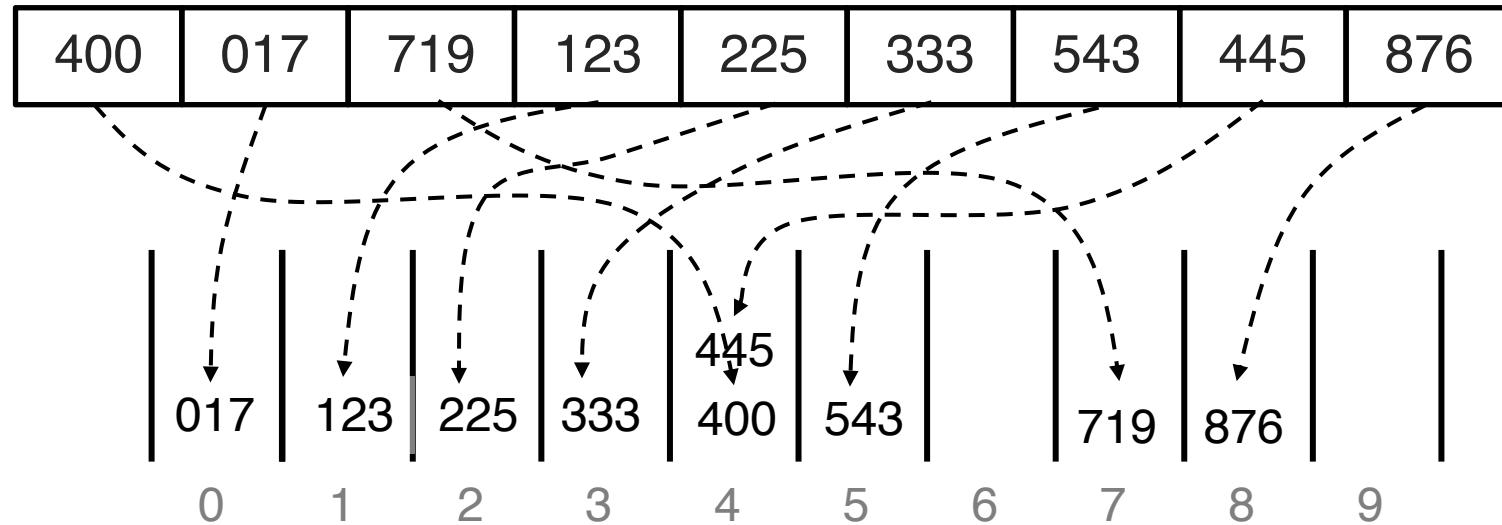
# Zweite Iteration (II)

Gehe Buckets von links nach rechts durch, entnimm Werte und füge Werte an nächster Stelle im Array ein:



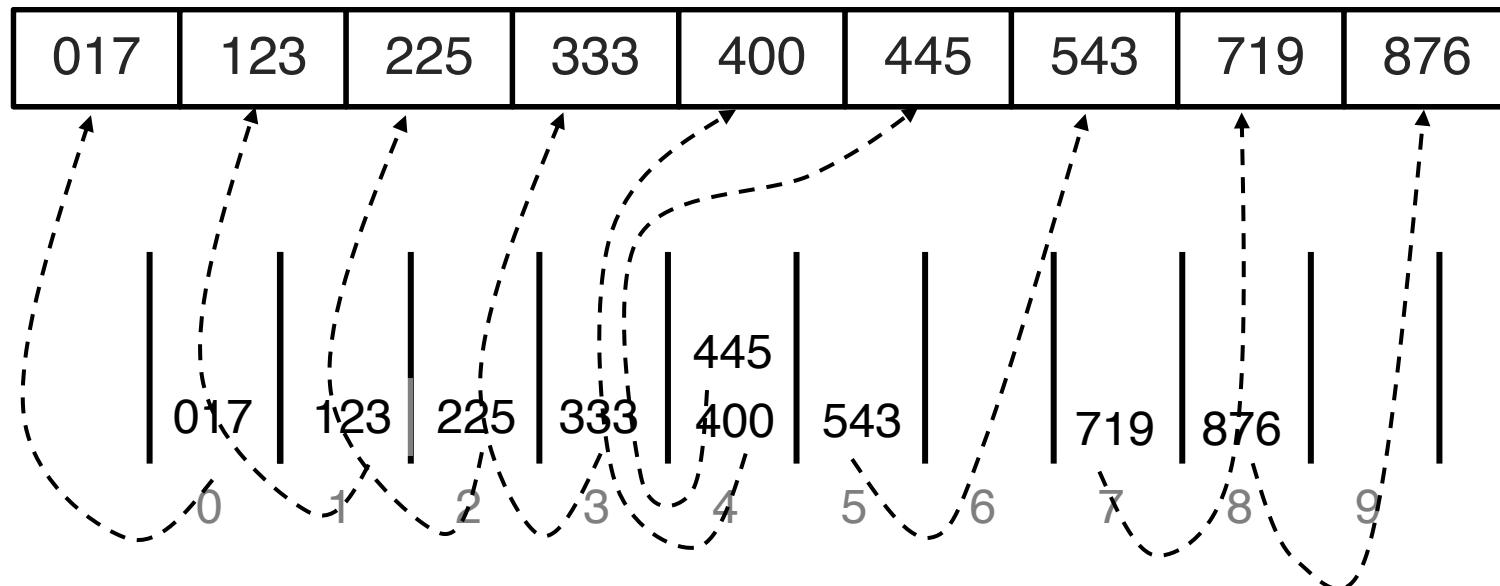
# Dritte Iteration (I)

Gehe Array von links nach rechts durch und füge Werte entsprechend **größtwertigster** Ziffer in Bucket ein:



# Dritte Iteration (II)

Gehe Buckets von links nach rechts durch, entnimm Werte und füge Werte an nächster Stelle im Array ein:



# Algorithmus

(Terminierung klar)

```
radixSort(A) // keys: d digits in range [0,D-1]
// B[0][],..., B[D-1][] buckets (init: B[k].size=0)

1  FOR i=0 TO d-1 DO //0 least, d-1 most sign. digit
2      FOR j=0 TO n-1 DO putBucket(A,B,i,j);
3      a=0;
4      FOR k=0 TO D-1 DO          //rewrite to array
5          FOR b=0 TO B[k].size-1 DO
6              A[a]=B[k][b]; //read out bucket in order
7              a=a+1;
8          B[k].size=0;        //clear bucket again
9  return A
```

```
putBucket(A,B,i,j) // call-by-reference
1  z=A[j].digit[i]; // i-th digit of A[j]
2  b=B[z].size;     // next free spot
3  B[z][b]=A[j];
4  B[z].size=B[z].size+1;
```

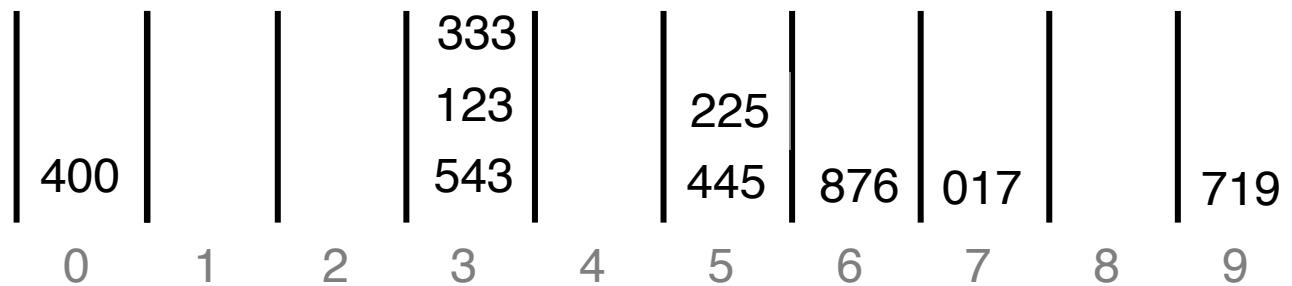
A.size=Anzahl einge-  
tragender Elemente in Array A

# Korrektheit (I)

Achtung: erste Iteration ( $i=1$ ) ist für Schleifenwert  $i=0$

Per Induktion: Nach  $i$ -ter Iteration ist Array gemäß letzten  $i$  Ziffern sortiert

400	543	123	333	445	225	876	017	719
-----	-----	-----	-----	-----	-----	-----	-----	-----

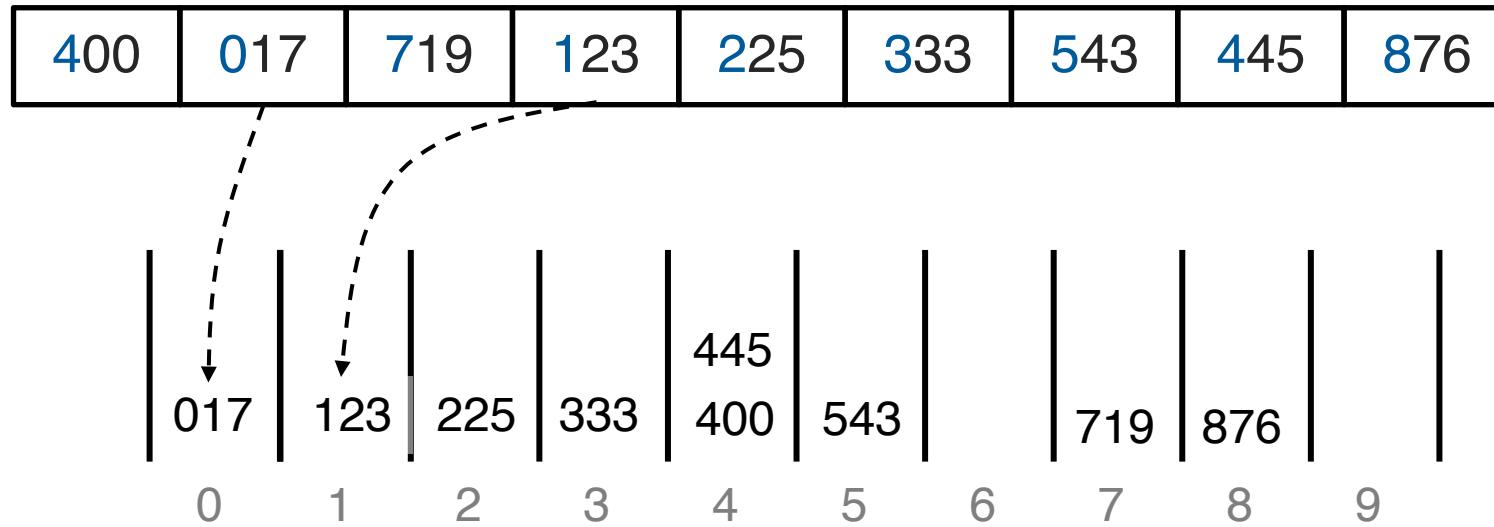


## Induktionsanfang $i=1$ :

Gilt nach erster Iteration, weil nach letzter Ziffer sortiert wird

# Korrektheit (II)

**Per Induktion:** Nach i-ter Iteration ist Array gemäß letzten i Ziffern sortiert

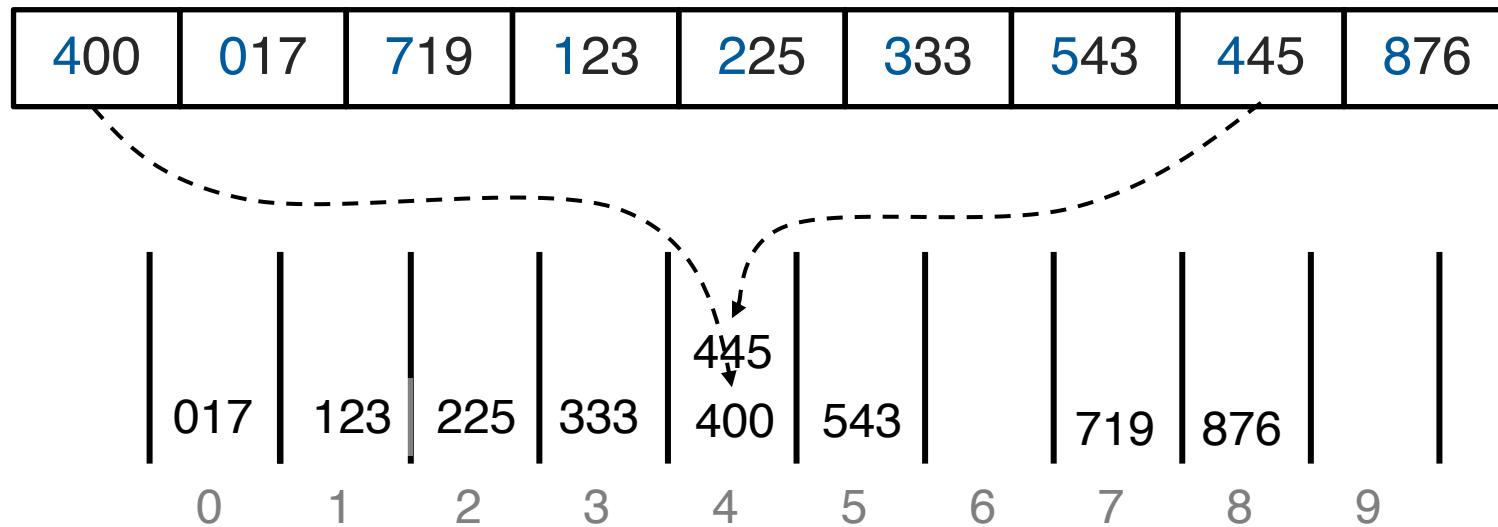


## Induktionsschritt $i \rightarrow i+1$ (1.Fall):

Wenn  $(i+1)$ -te Ziffer zweier beliebiger Werte verschieden, dann wird Wert mit kleinerer  $(i+1)$ -ten Ziffer weiter vorne einsortiert, zuerst wieder ausgelesen und steht somit auch im Array vorher

# Korrektheit (III)

**Per Induktion:** Nach i-ter Iteration ist Array gemäß letzten i Ziffern sortiert  
**(und damit nach d-ter Iteration das Array sortiert)**



## Induktionsschritt $i \rightarrow i+1$ (2. Fall):

Wenn  $(i+1)$ -te Ziffer gleich, dann steht nach Induktionsvoraussetzung der auf letzten  $i$  Ziffern kleinere Wert weiter links, wird zuerst einsortiert und auch zuerst wieder ausgelesen

# Laufzeit

```
radixSort(A) // keys: d digits in range [0,D-1]  
// B[0][], ..., B[D-1][] buckets (init: B[k].size=0)
```

```
1  FOR i=0 TO d-1 DO // 0 least, d-1 most sign. digit  
2      FOR j=0 TO n-1 DO putBucket(A,B,i,j); _____ O(n)  
3      a=0;  
4      FOR k=0 TO D-1 DO          // rewrite to array  
5          FOR b=0 TO B[k].size-1 DO  
6              A[a]=B[k][b]; // read out bucket in order  
7              a=a+1;  
8          B[k].size=0;        // clear bucket again _____ O(D)  
9  return A
```

Gesamtlaufzeit  
 $\mathcal{O}(d \cdot (n + D))$

$\mathcal{O}(n)$   
Schritte  
(alles in A kopieren)

$\mathcal{O}(D)$

```
putBucket(A,B,i,j) // call-by-reference  
1  z=A[j].digit[i]; // i-th digit of A[j]  
2  b=B[z].size;     // next free spot  
3  B[z][b]=A[j];  
4  B[z].size=B[z].size+1;
```

$O(1)$

# Laufzeit – Interpretation

Gesamlaufzeit  
 $\mathcal{O}(d \cdot (n + D))$

Größe Zifferbereich  $D$  oft als konstant angesehen:

Laufzeit  $\mathcal{O}(dn)$

Wenn auch  $d$  als konstant angesehen:

Laufzeit  $\mathcal{O}(n)$

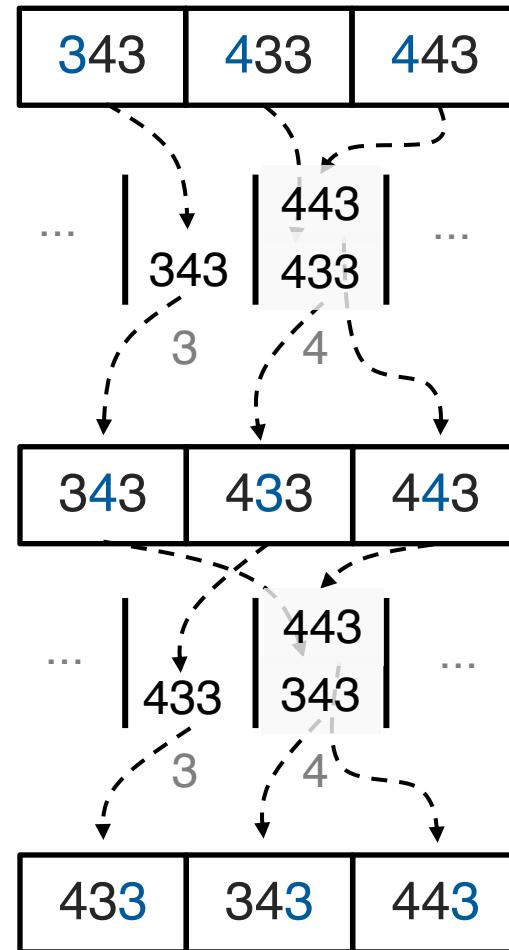
linear!

Aber:

eindeutige Schlüssel für  $n$  Elemente  
benötigen  $d = \Theta(\log_D n)$  Ziffern!

Laufzeit  $\mathcal{O}(n \cdot \log n)$

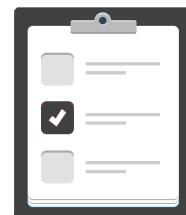
# Mit höchstwertiger Ziffer beginnen?



...folgende Iteration ändert Reihenfolge nicht mehr



Wo scheitert der Korrektheitsbeweis beim Sortieren beginnend mit der höchstwertigen Ziffer?



Was sind Beispiele von Schlüsselwerten, die sich nicht per RadixSort sortieren lassen?



Ein Sortieralgorithmus ist stabil, wenn es die relative Ordnung von gleichen Werten beibehält.  
Überlegen Sie sich, dass RadixSort stabil ist.