Algorithmen und Datenstrukturen

Marc Fischlin, SS 2024

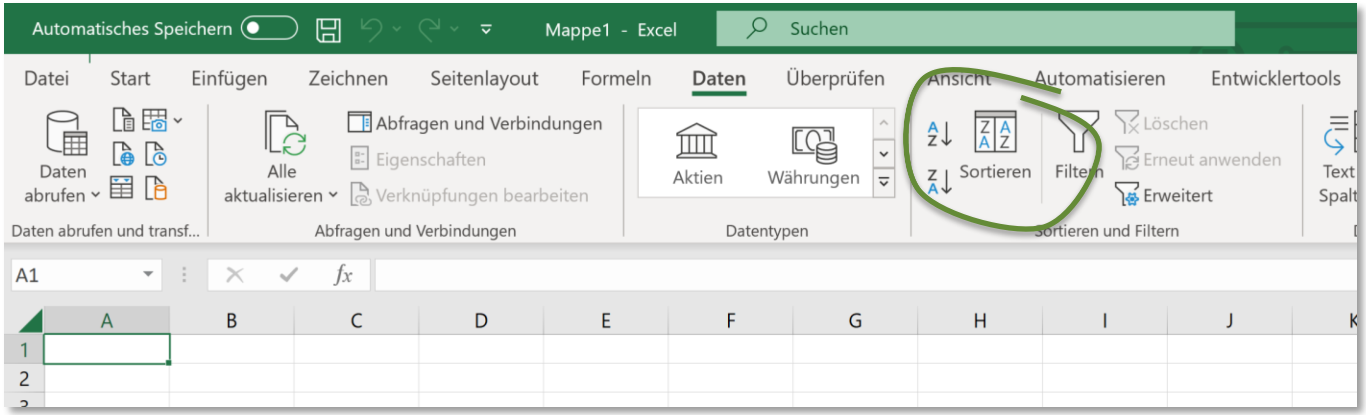
02

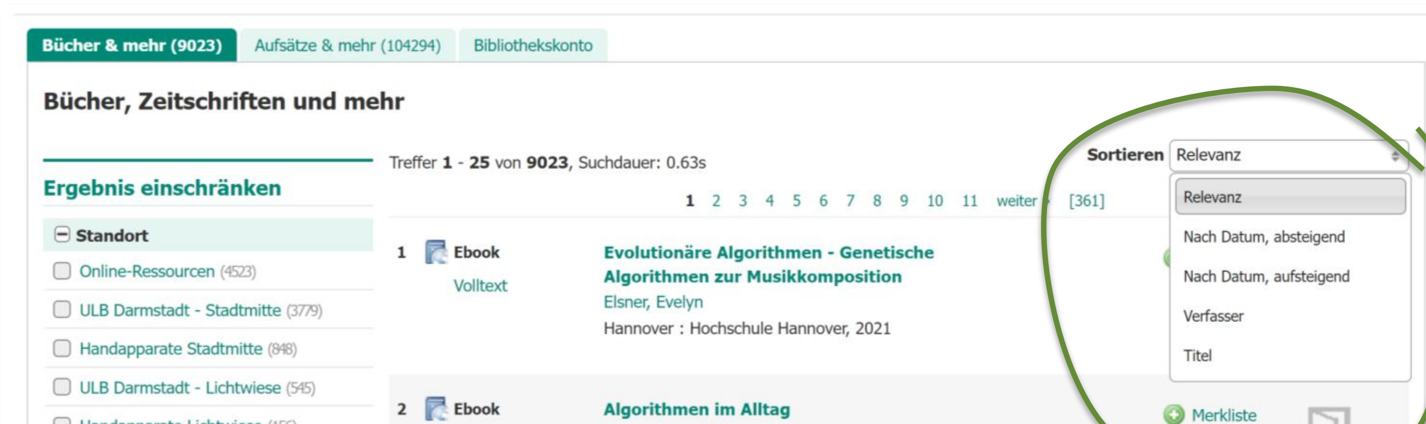
Sortieren

Folien beruhen auf gemeinsamer Veranstaltung mit Christian Janson aus dem SS 2020

Das Sortierproblem

Sortierproblem in der Praxis





Sortierproblem

Gegeben: Folge von Objekten

Gesucht: Sortierung der Objekte
(gemäß ausgewiesenen Schlüsselwerts)

Schlüssel(wert)
„Satellitendaten“

Objekt

Schlüssel(wert): Wert, nach dem Objekt sortiert wird

Beispiel

Steuer-ID
Name Adresse Bankverbindung …

Schlüsselproblem

Schlüssel(werte) müssen nicht eindeutig sein,

aber müssen „sortierbar“ sein

Schlüssel(wert) — Wert, nach dem Objekt sortiert wird

„Satellitendaten“

Objekt

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

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

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

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

(ohne Totalität: partielle Ordnung)

Bemerkung: Totale Ordnung $\leq$ impliziert Relation $>$ durch $x > y :\Leftrightarrow x \nleq y$

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

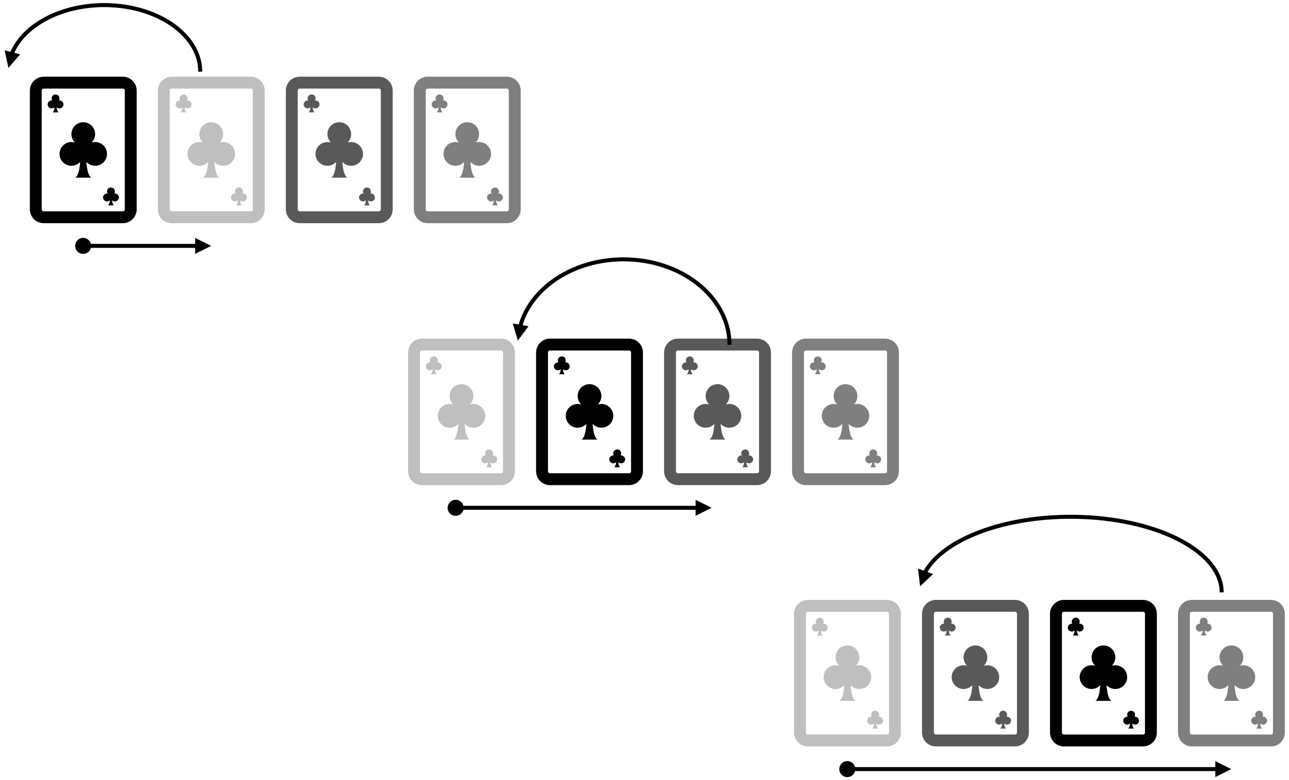
`A.length` beschreibt (fixe) Länge des Arrays, im Beispiel 9

`A[i…j]` beschreibt Teil-Array von Index **`i`** bis **`j`**, **`0=<i=<j=<A.length-1`**

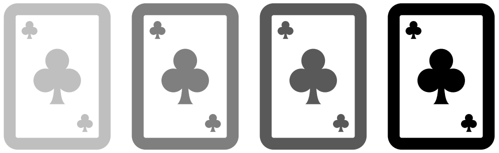
Insertion Sort

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)
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;
```

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

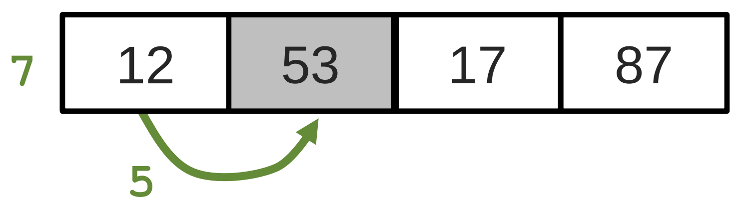
Short Circuit Evaluation (wie in Java):
Wenn erste AND-Bedingung **false**, wird zweite Bedingung nicht mehr ausgewertet

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



6 j=-1

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`

12	53	53	87

5

`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 < key=17`

12	17	53	87

7

`6 j=0`

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)
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;
```

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)
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;
```

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]`**.

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],…,A[n-1]=a[n-1]`**.

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]`**.

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],…,A[i-2]`** sortierte Version von **`a[0],…,a[i-2]`** und **`A[i-1]=a[i-1],…,A[n-1]=a[n-1]`**

Durch die **`WHILE`**-Schleife wurde **`A[i-1]=a[i-1]`** nach links einsortiert und größere Elemente von **`A[0],…,A[i-2]`** um jeweils eine Position nach rechts verschoben. Elemente **`A[i],…,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],…,A[i-1]`** den sortierten ursprünglichen Eingabewerten **`a[0],…,a[i-1]`**. Ferner gilt **`A[i]=a[i],…, 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],…,A[n-1]` ist sortierte Version von **`a[0],…,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],…,A[i-1]`** den sortierten ursprünglichen Eingabewerten **`a[0],…,a[i-1]`**. Ferner gilt **`A[i]=a[i],…, 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:
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
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	
5	c5	
6	c6	
7	c7	$n-1$

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 ba
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 ba
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:

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

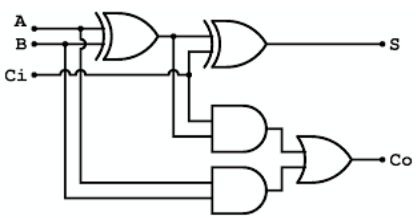
Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile i hat Aufwand ci

Kosten für individuelle Schritte?

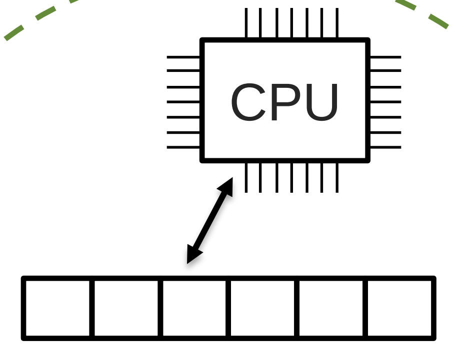
Wie teuer ist z.B. Zuweisung **`A[j+1]=A[j]`** in Zeile **`5`**, also was ist $c5$?

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

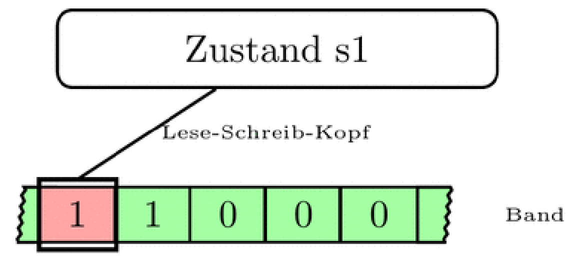


Quelle: ke.tu-darmstadt.de

Schaltkreis



RAM



Quelle: de.wikipedia.org/wiki/Turingmaschine

Turingmaschine

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

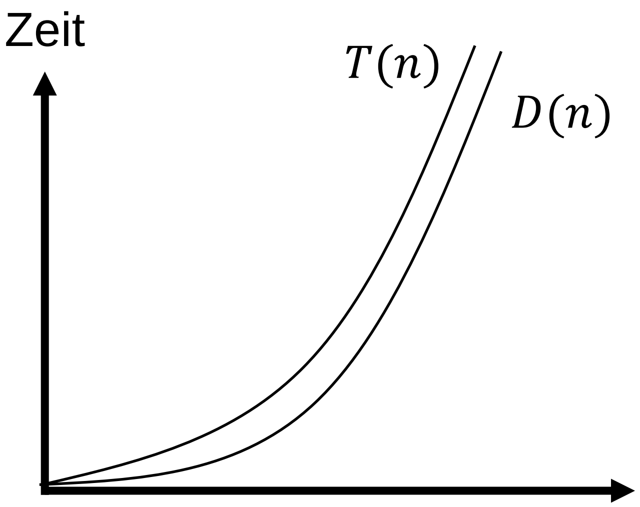
Asymptotische Vereinfachung (I)

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** $\boldsymbol{(T(n) - D(n))/T(n)}$
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) :

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

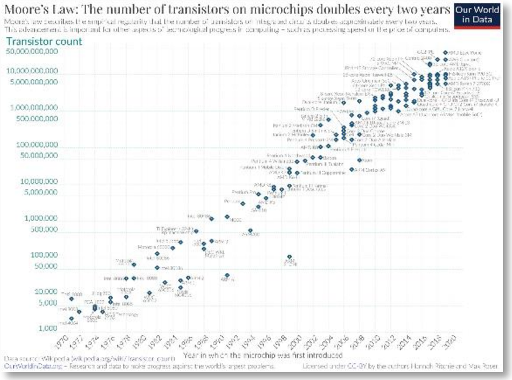
Zum Vergleich (nur dominanter Term) :

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

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/Mooresches_Gesetz`

Θ-Notation / Landau-Symbole

Paul Bachmann
Edmund Landau
ca. 1900

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

Eingabekomplexität Laufzeit

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

Positive Konstanten

$f(n)$ wird von $c_1 g(n)$ und $c_2 g(n)$ für hinreichend große n eingeschlossen

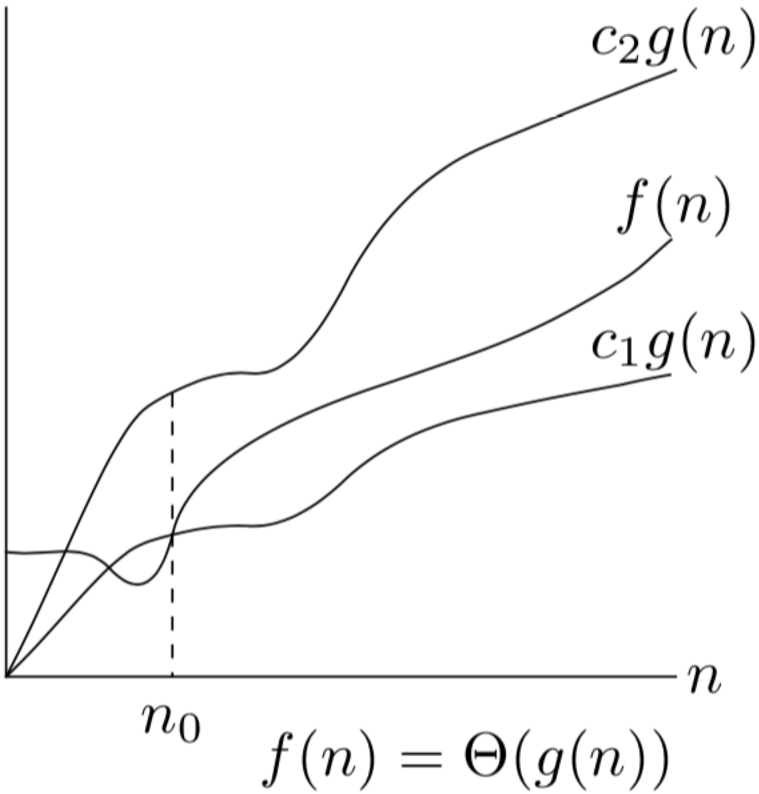
Funktion f

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)

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

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

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

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

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

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

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

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

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

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

O-Notation

Obere asymptotische Schranke

$$O(g) = \{f : \underbrace{\exists\, c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}}_{\text{Positive Konstanten}}, \forall n \geq n_0, \underbrace{0 \leq f(n) \leq cg(n)}_{f(n) \text{ wird von } cg(n) \text{ für hinreichend große } n \text{ beschränkt}}\}$$

Funktion f

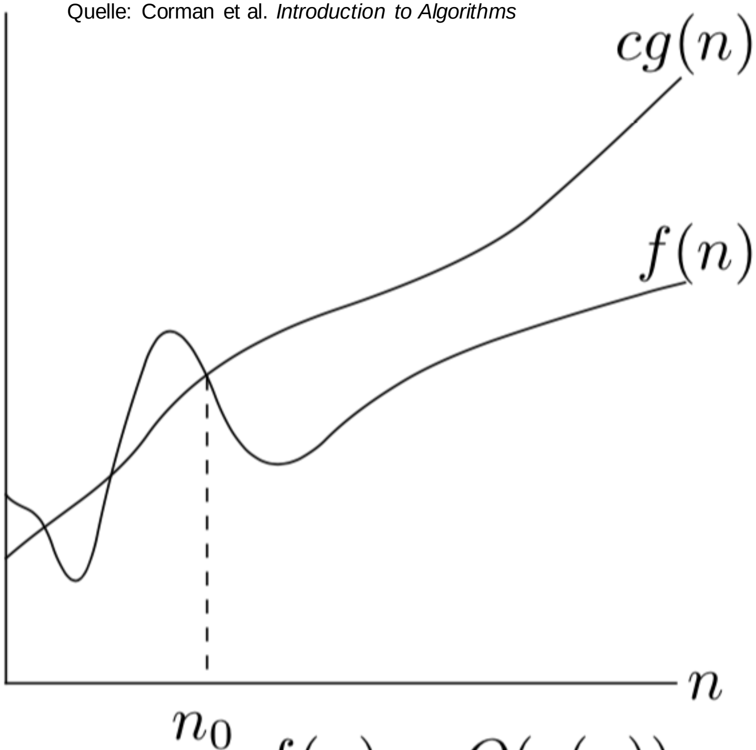
Für alle n größer gleich n_0

Sprechweise: f wächst höchstens so schnell wie g

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

Veranschaulichung der 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: $\Theta(g(n)) \subseteq \mathrm{O}(g(n))$ und somit $f(n) = \Theta(g) \Longrightarrow f(n) = \mathrm{O}(g)$

O-Notation: Rechenregeln

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

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

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

punktweise

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

Ω-Notation

Untere asymptotische Schranke

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

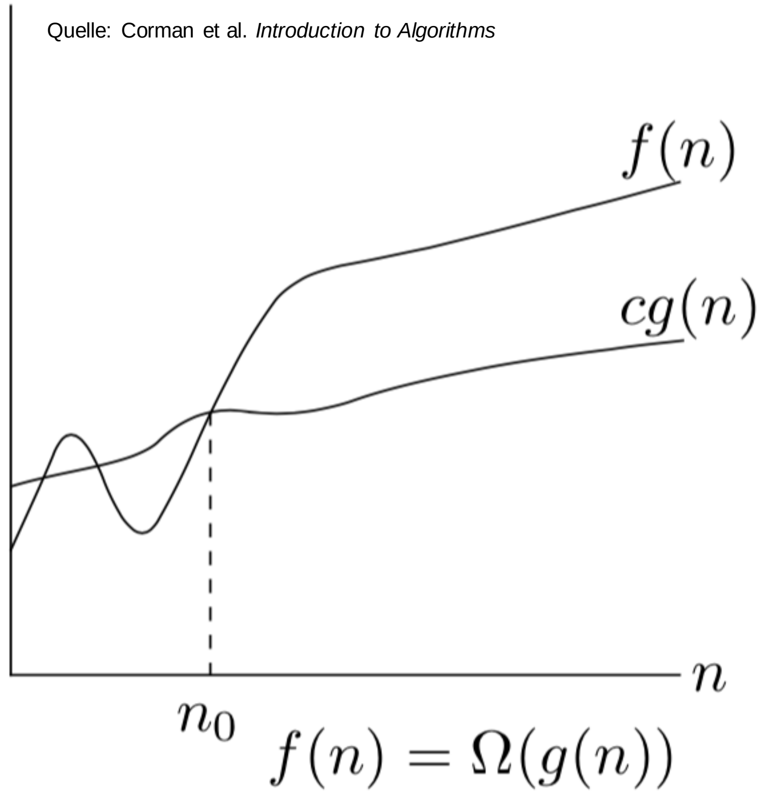
Funktion f

Für alle n größer gleich n_0

Sprechweise: f wächst mindestens so schnell wie g

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

Veranschaulichung der Ω-Notation



$n \geq n_0$:

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

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 O-Notation.

Überlegen Sie sich, dass gilt:
$O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(n^4) \subseteq O(n^5) \subseteq \ldots$
und
$\Omega(n) \supseteq \Omega(n^2) \supseteq \Omega(n^3) \supseteq \Omega(n^4) \supseteq \Omega(n^5) \supseteq \ldots$
und dass die Inklusionen jeweils strikt sind

Zusammenhang O, Ω und Θ

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

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

genau dann, wenn

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

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

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

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

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

Anwendung O-Notation (I)

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

Allgemein besser immer als Mengen auffassen
und von links nach rechts lesen (mit $\in, \subseteq$):

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

$$\in \qquad\qquad\qquad \subseteq \qquad\qquad \subseteq$$

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

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

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

Anwendung O-Notation (II)

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

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

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

O, Ω 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 $\leq T(n) = O(n^2)$

O, Ω 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

O, Ω 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:

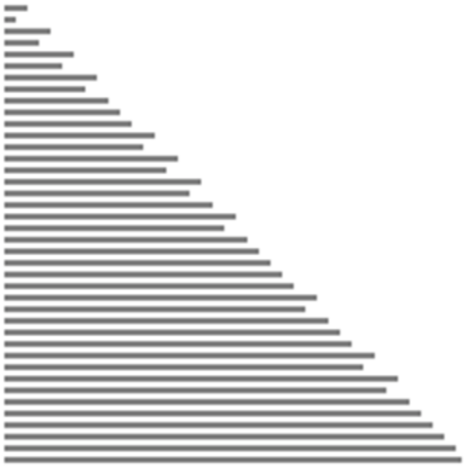
A	1	2	3		$\dots$	$\dots$		$n-2$	$n-1$

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

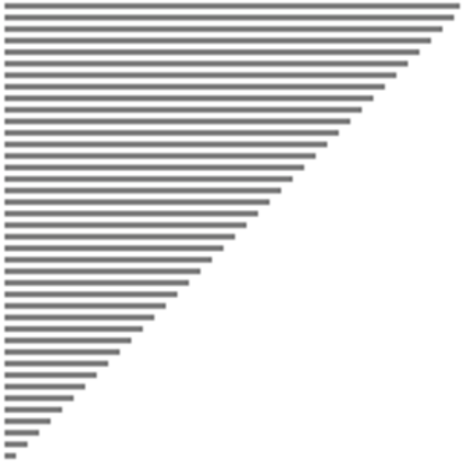
Insgesamt macht Algorithmus für dieses **A** also $\Theta(n)$ Schritte

Laufzeit Insertion Sort (I)

„guter“ Fall
(fast vorsortiertes Array)



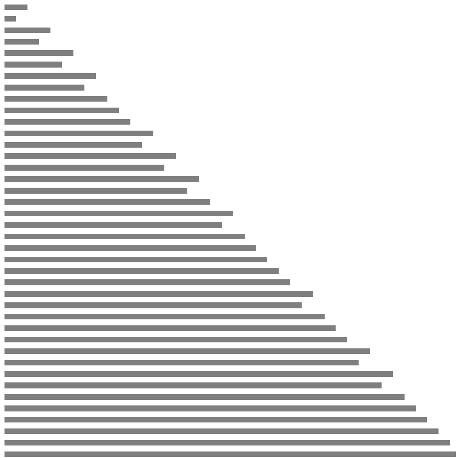
„schlechter“ Fall
(invertiertes, vorsortiertes Array)



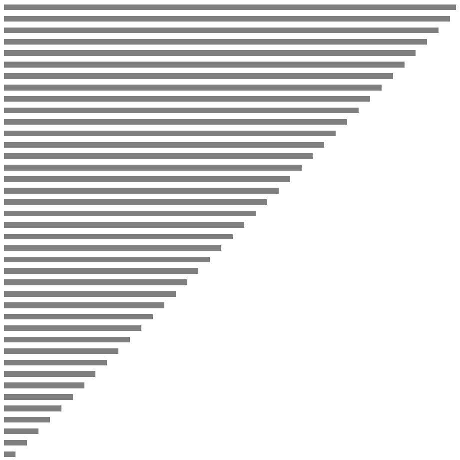
Quelle: **`https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/insertion-sort`**

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	Matrixmultiplikation
$\Theta(n^k)$	Polynomiell	
$\Theta(2^n)$	Exponentiell	Travelling-Salesperson*
$\Theta(n!)$	Faktoriell	Permutationen

*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 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?

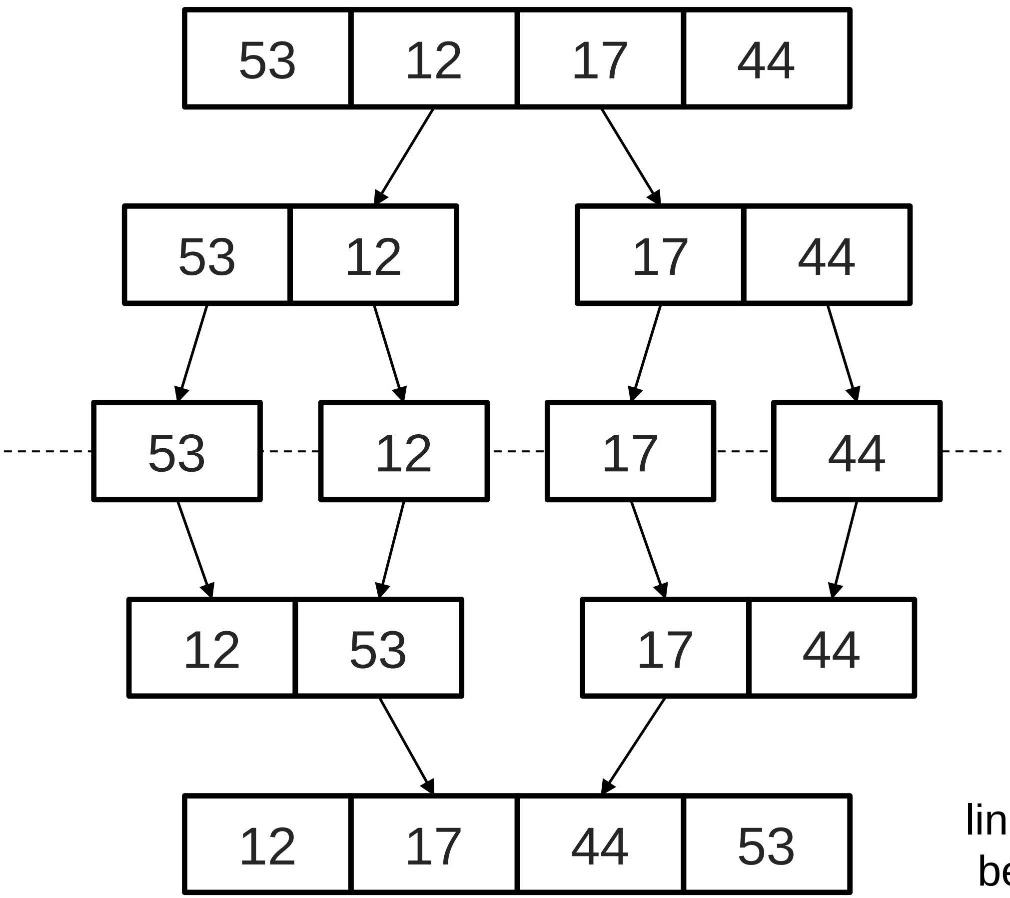
Merge Sort

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:

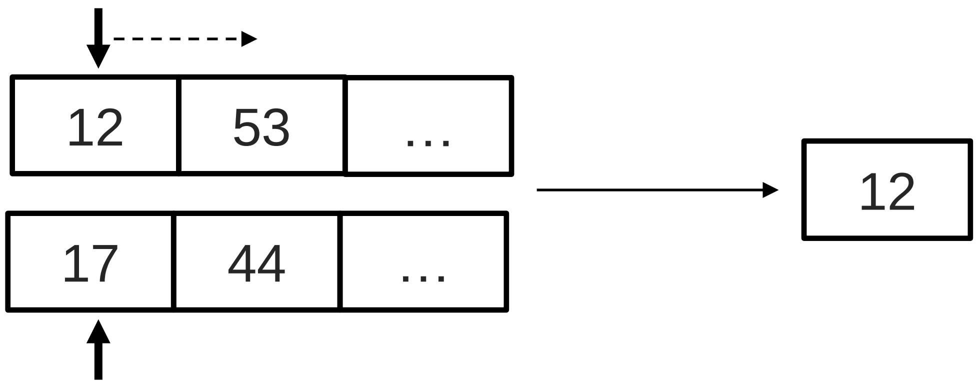
left=3, right=4, mid=3

left=3, right=5, mid=4

Merge (für sortierte Teillisten)

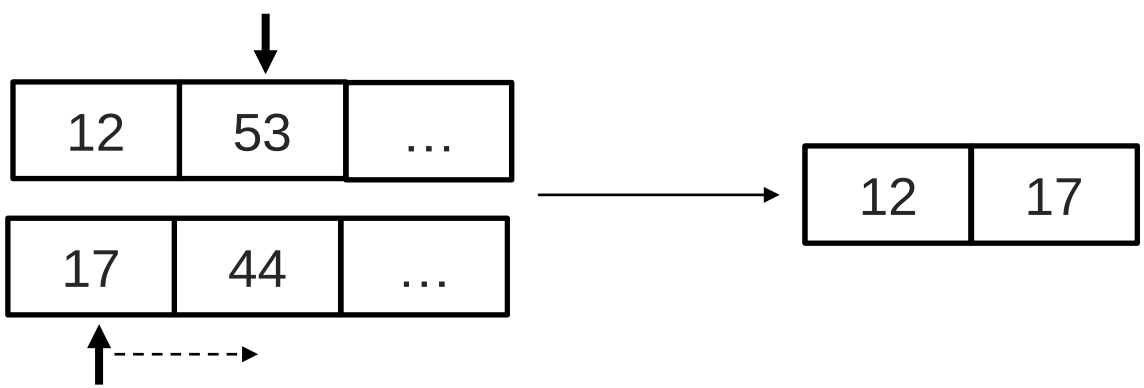
Idee : nimm nächstes kleinstes Element aus linker oder rechter Teilliste und gehe in dieser Liste eine Position nach rechts

(1)



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

(2)



…

Algorithmus: Merge (für sortierte Teillisten)

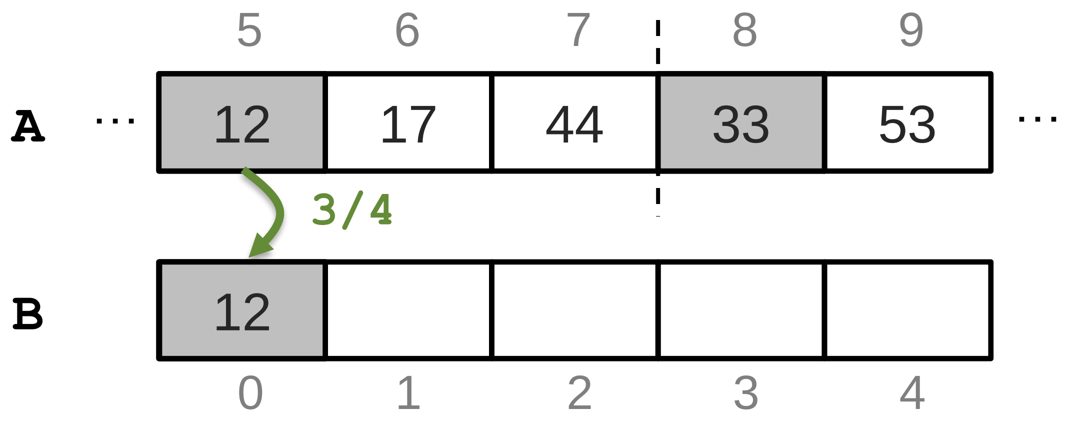
rechte Liste noch aktiv und
[linke Liste bereits abgearbeitet oder
nächstes Element rechts]

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`

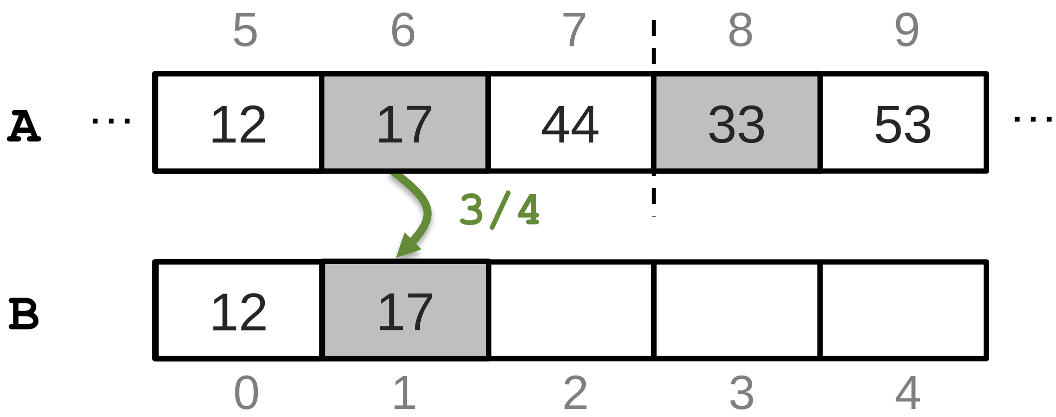


`p=5, q=8, i=0`

```
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)

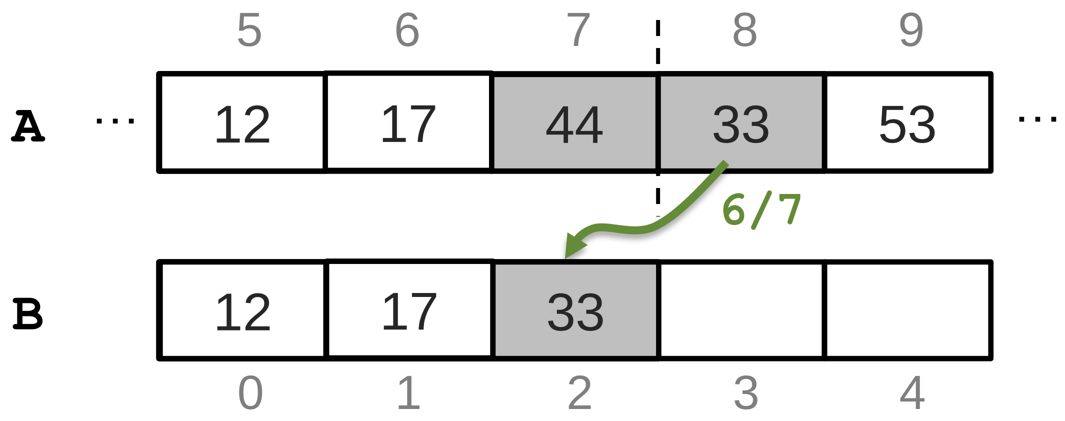
`left=5, mid=7, right=9`



`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`

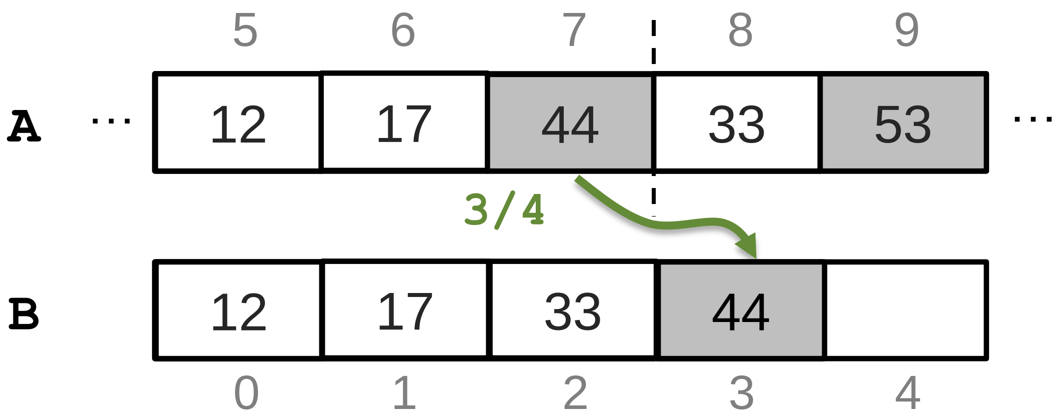


`p=7, q=8, i=2`

```
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`

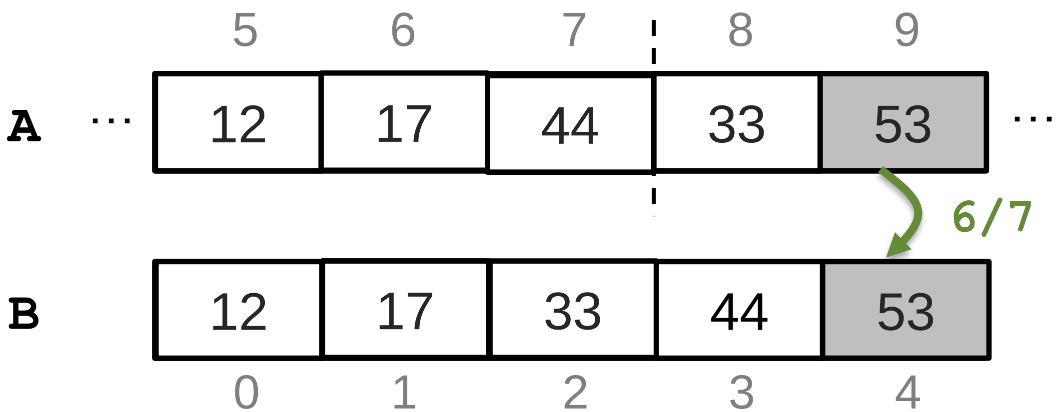


`p=7, q=9, i=3`

```
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	…

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

	0	1	2	3	4
B	12	17	33	44	53

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]=`$-\infty$

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**

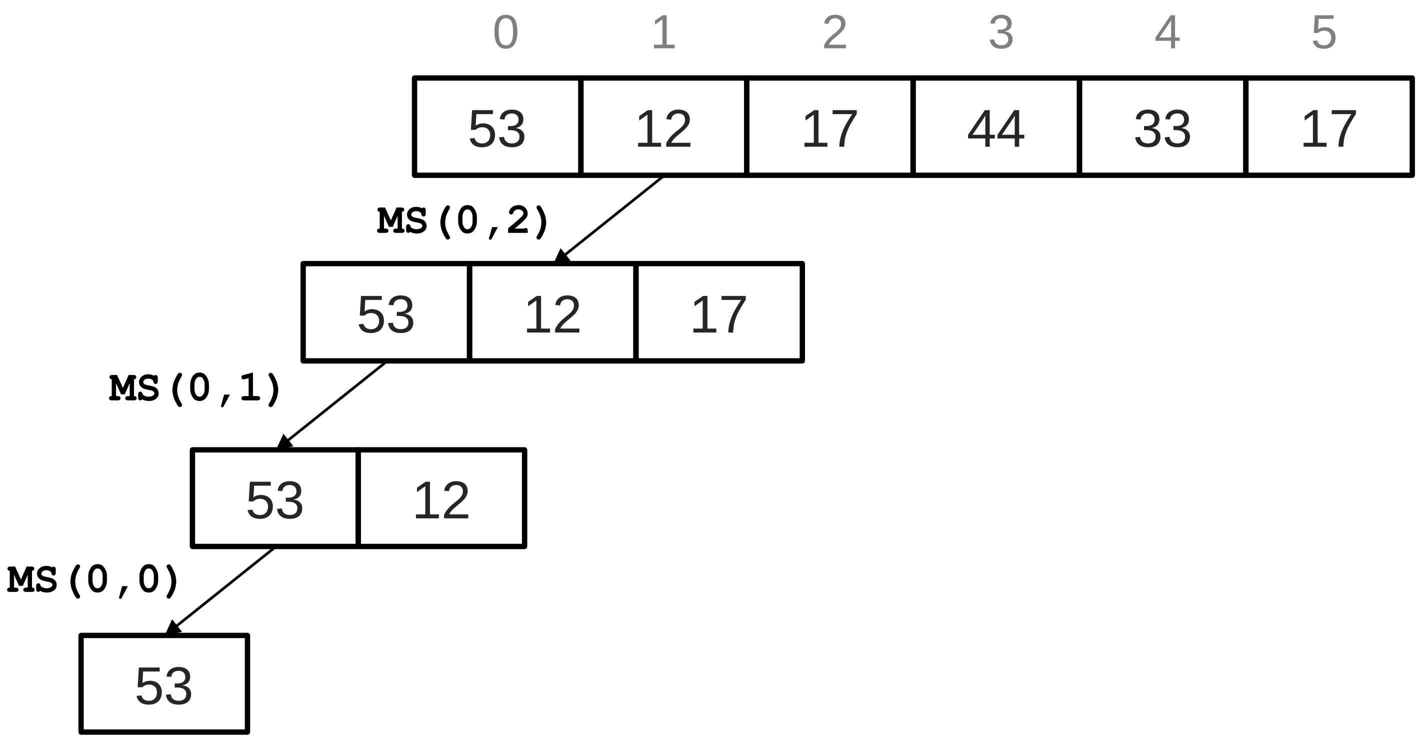
Also besteht **B[0…right-left]** aus **A[left…mid],A[mid+1…right]**
und ist sortiert

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
```

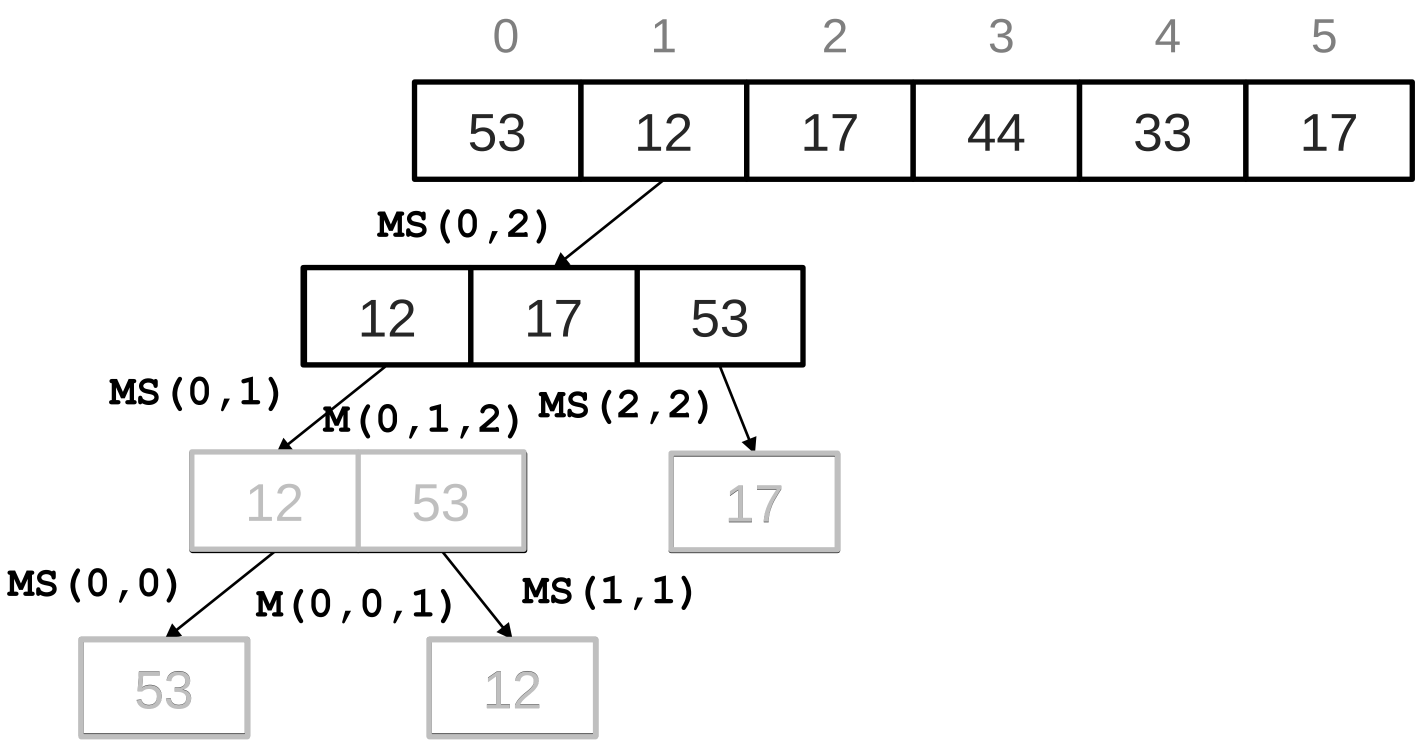
Beispiel: Merge Sort (I)

```
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
```



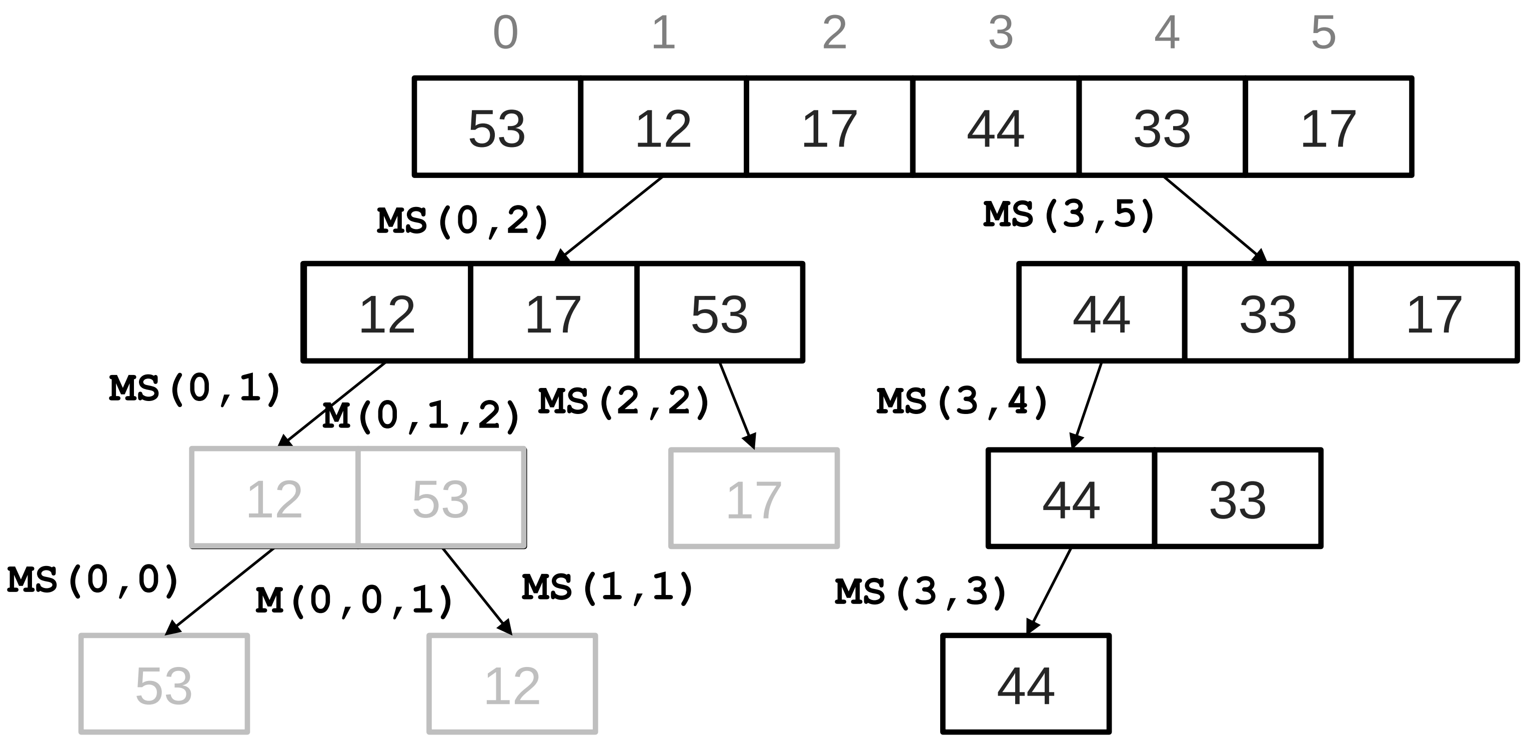
Beispiel: Merge Sort (II)

```
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
```



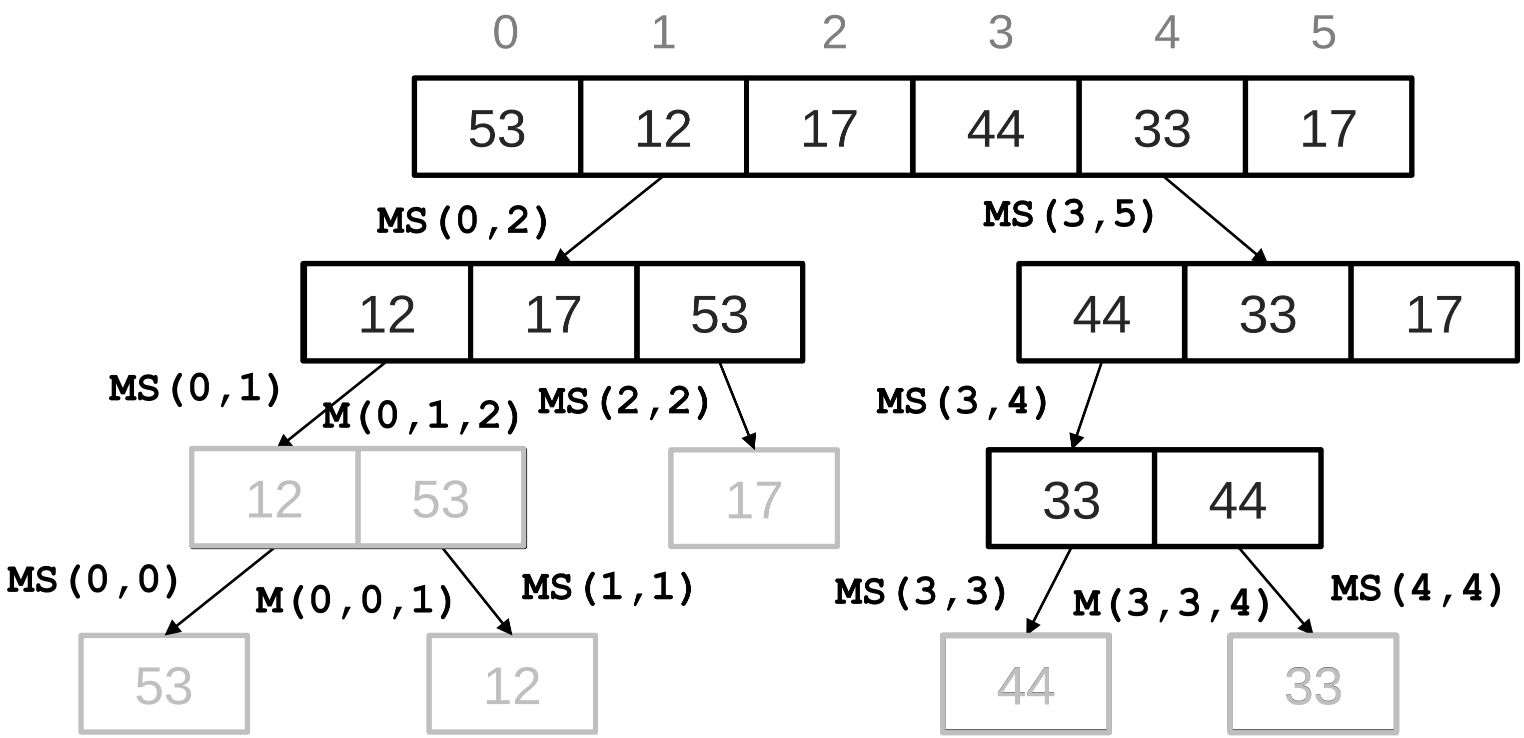
Beispiel: Merge Sort (III)

```
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
```



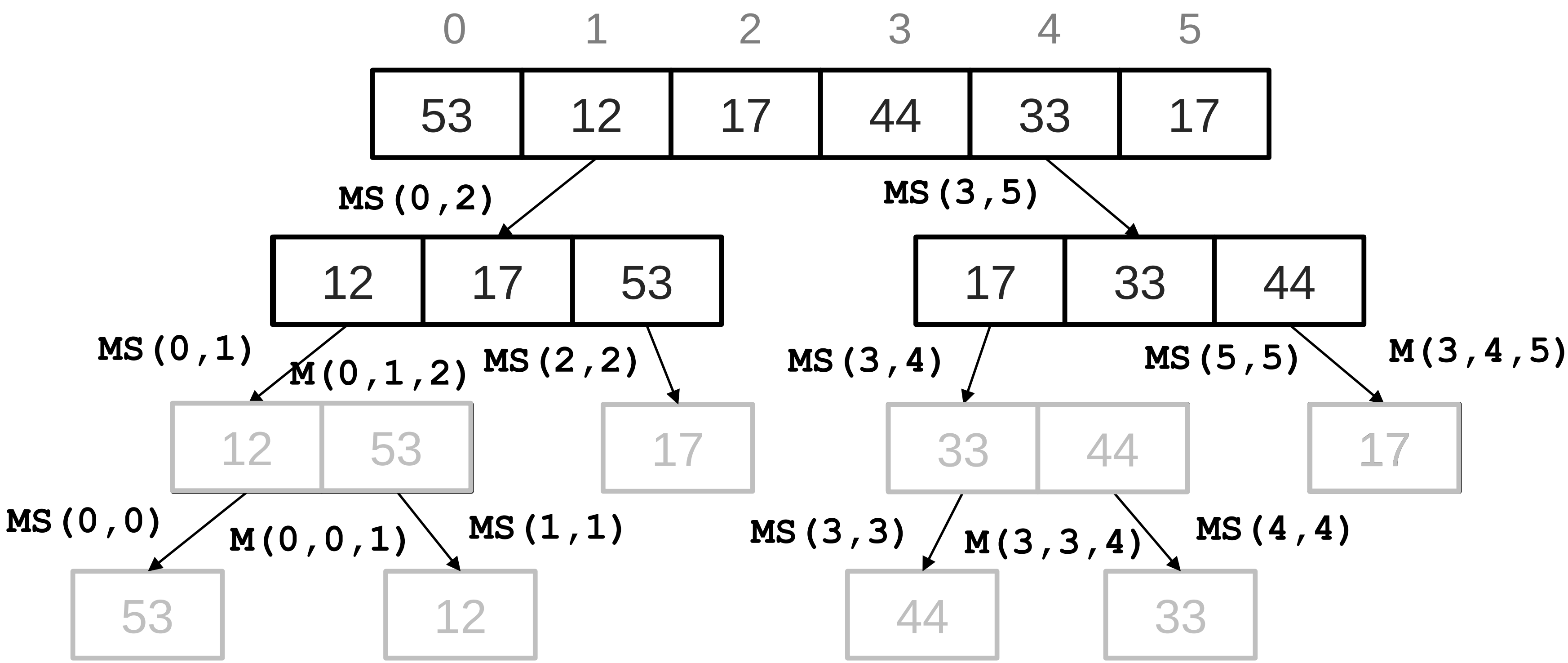
Beispiel: Merge Sort (IV)

```
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
```



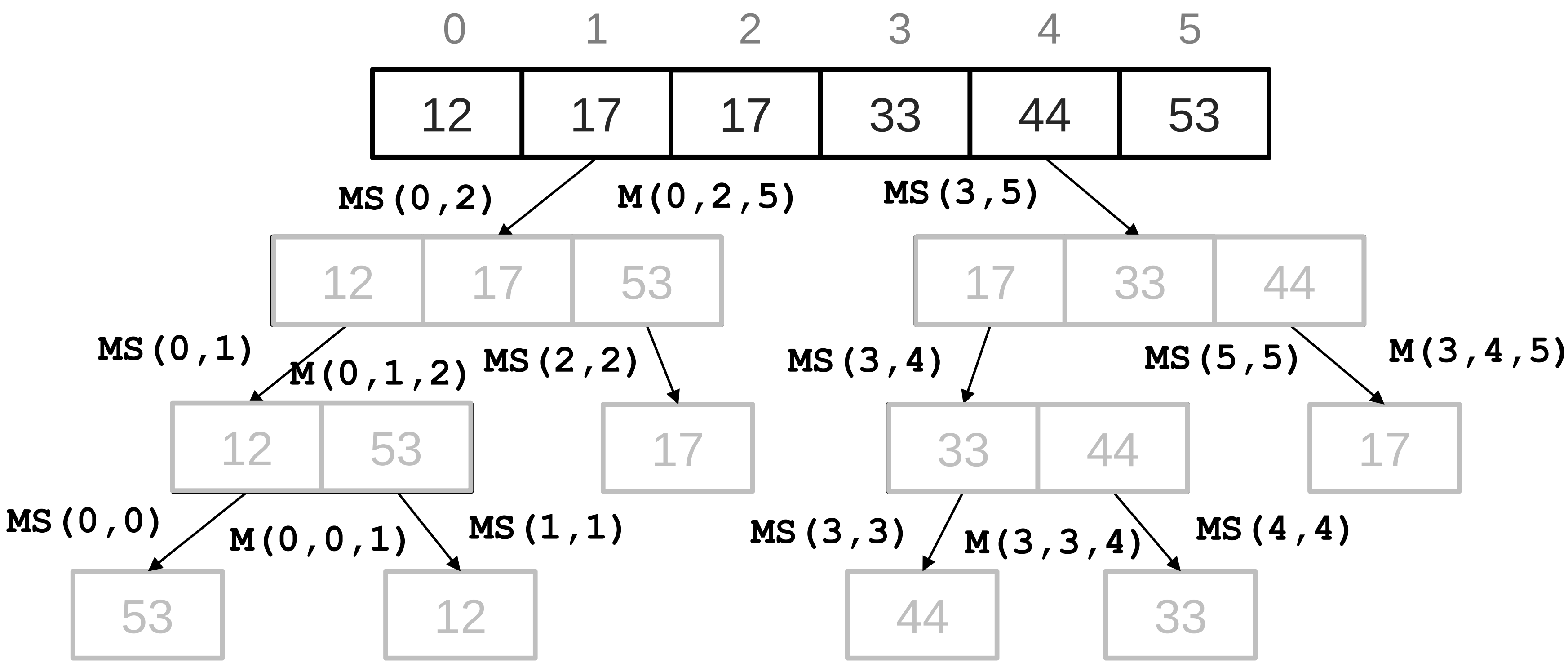
Beispiel: Merge Sort (V)

```
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:

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

zwei rekursive Aufrufe für jeweils halb so großes Array

`IF` + `floor` (konstanter Aufwand)

`Merge` (Aufwand $O(n)$)

(ignorieren hier zur Vereinfachung Runden)

und $T(1) \leq c$, weil dann **`low=hi`**

Rekursion „manuell iterieren“

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

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

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

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

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

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

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

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 $af(n/b) \le cf(n)$ 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 $\Big\downarrow$ $a = 4, b = 3,\ \log_b a = 1.26\ldots, \varepsilon = 0.26\ldots$
$f(n) = \Theta(n) = \Theta(n^{\log_b(a)-\varepsilon})$

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

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

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

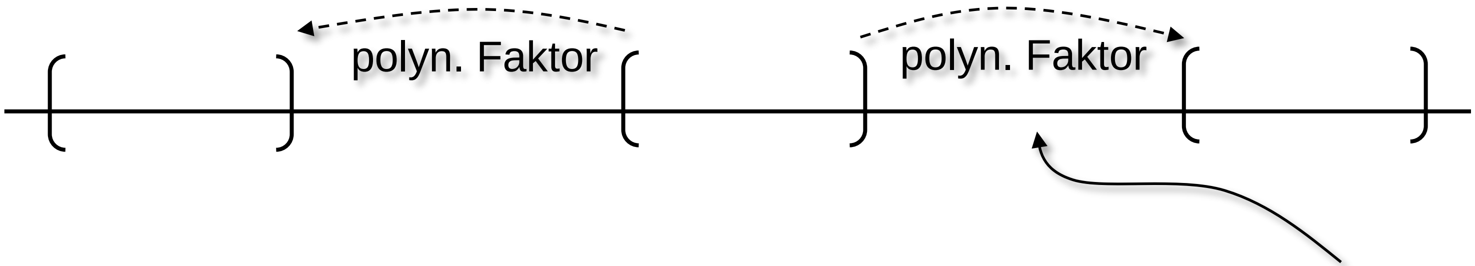
$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

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

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



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

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

---

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

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

Lösen Sie folgende Rekursionsgleichung

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

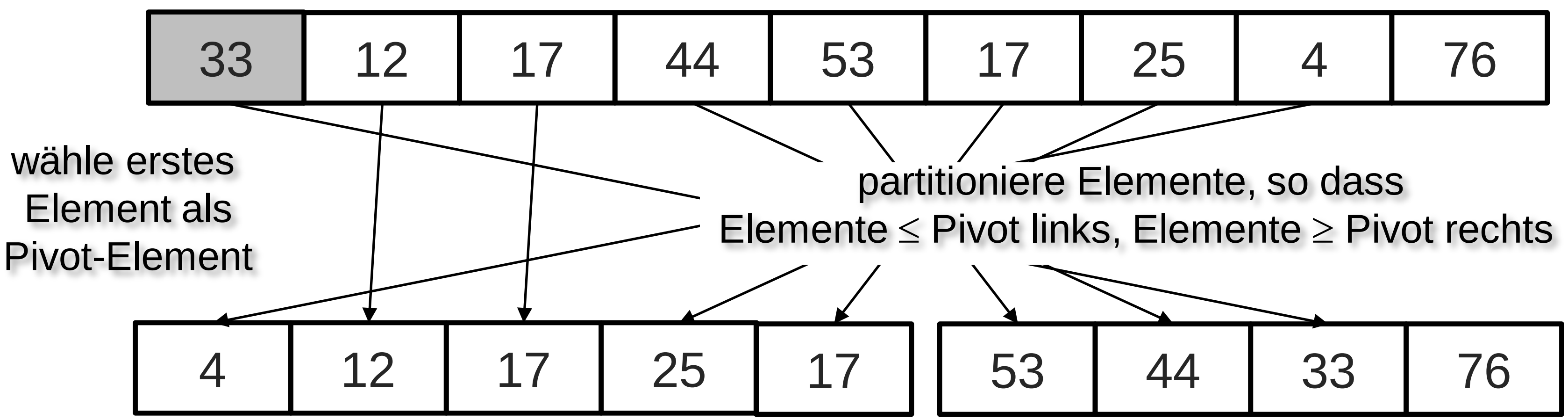
mit Hilfe des Mastertheorems.

Quicksort

Idee

wie in Merge Sort verwendet Quicksort Divide & Conquer-Ansatz

Quicksort steckt mehr Arbeit in Aufteilen, Zusammenfügen kostenlos



sortiere beide Teil-Arrays rekursiv

Ergebnis ist komplett sortiertes Array

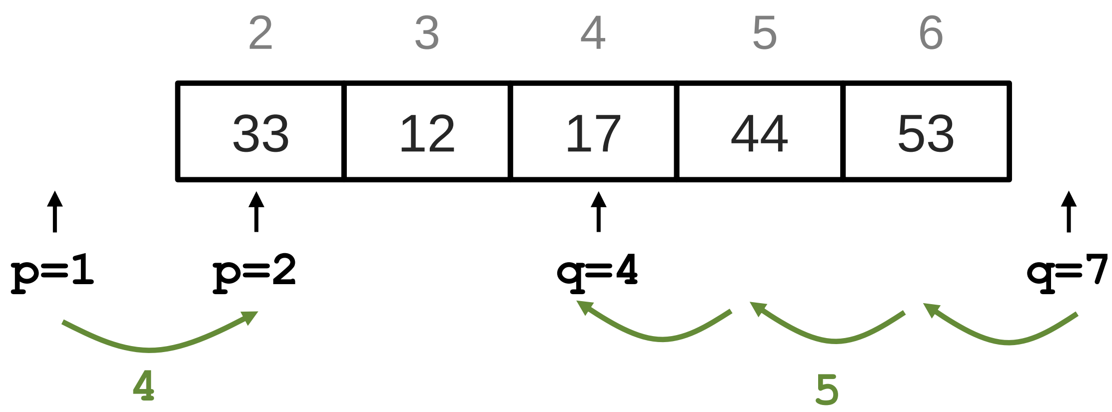
Algorithmus: Quicksort

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

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

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

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

Beispiel #1: Partition (I)

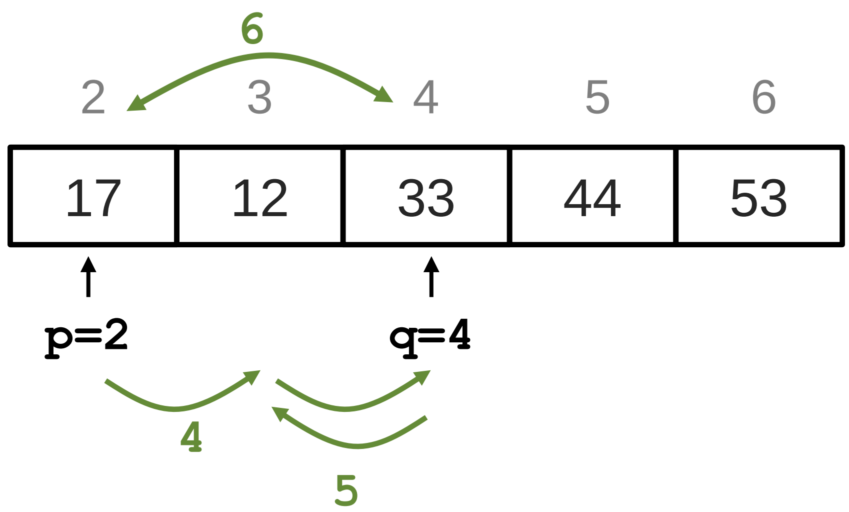
```
pivot=33
left=2, right=6
```




```
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

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

Beispiel #1: Partition (III)

pivot=33
left=2, right=6

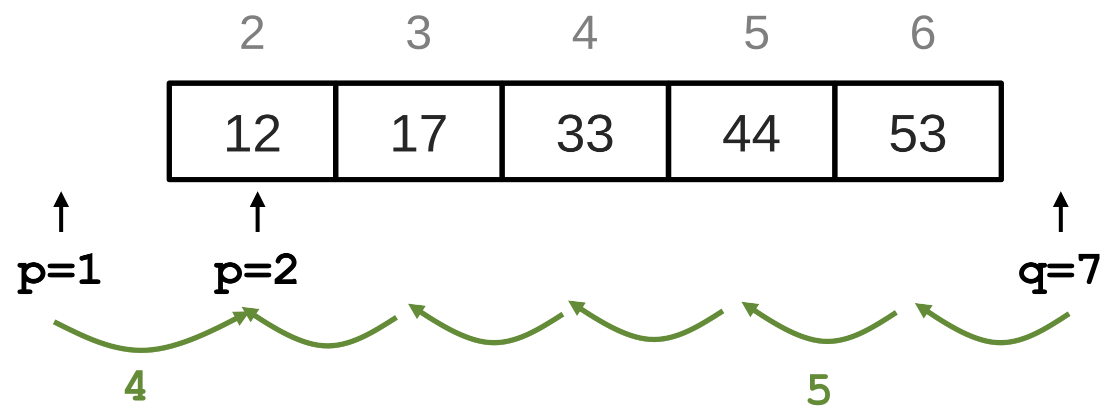
2	3	4	5	6
17	12	33	44	53
	q=3	p=4		

7 return 3

```
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
```

2	3	4	5	6
12	17	33	44	53

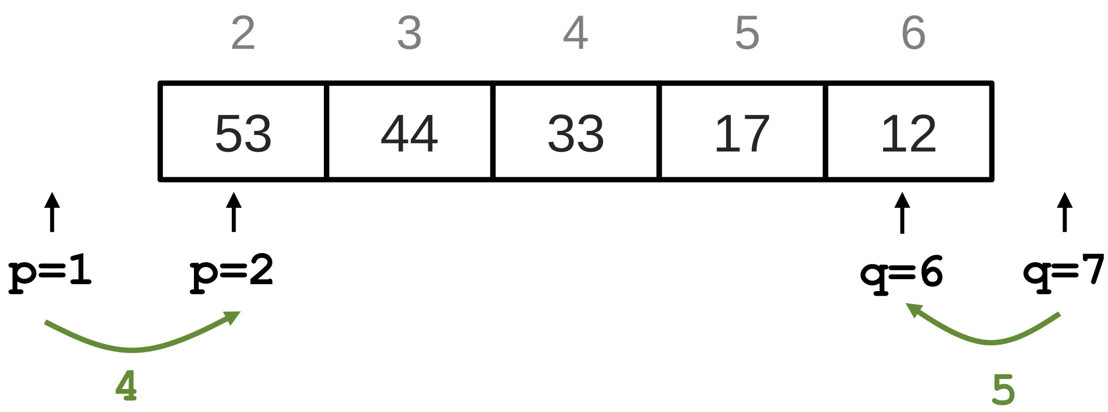
p=q=2

7 **return 2**

```
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)

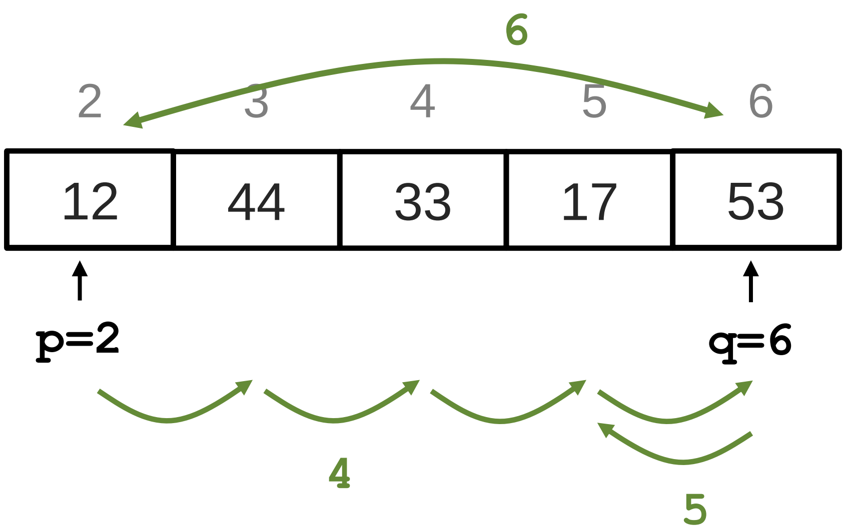
`pivot=53`
`left=2, right=6`




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

pivot=A[left];
p=left-1; q=right+1;  //move from left resp. right
WHILE p<q DO
    REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
    REPEAT q=q-1 UNTIL A[q]=<pivot; //right down
    IF p<q THEN Swap(A[p],A[q]);
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
```

2	3	4	5	6
12	44	33	17	53

q=5   p=6

```
7 return 5
```

```
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 ≥ **`pivot`**

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

2	3	4	5	6
33	12	53	17	44

`pivot=33`

↑
`p=left-1`

Partition Terminierung (II)

Betrachte **REPEAT**-Schleife **4**

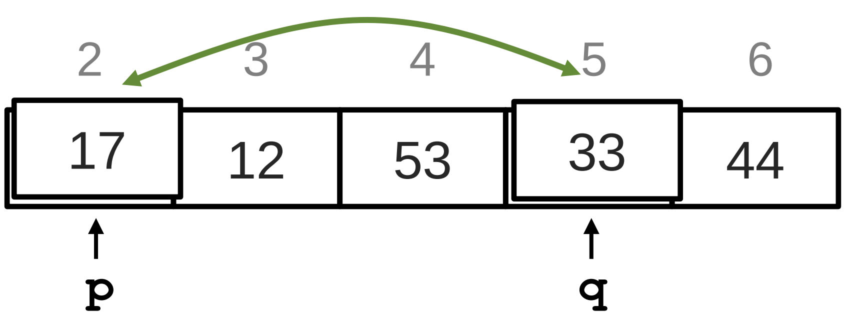
Behauptung: Vor Eintritt in **4** steht rechts von **p** stets Element ≥ **pivot**

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

Dann zeigt **p** nach **REPEAT** in voriger **WHILE**-Iteration auf Wert **>=pivot**

In aktueller Iteration wird **REPEAT** nur erneut erreicht, wenn **p<q** und somit in voriger **WHILE**-Iteration **Swap** ausgeführt wurde

Swap tauschte **A[p]>=pivot** weiter nach rechts (**p<q**), daraus folgt Behauptung



pivot=33

Partition Terminierung (III)

Analog: Vor Eintritt in **5** steht links von **q** stets ein Element ≤ **pivot**

Folglich terminieren beide **REPEAT**s 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]
```

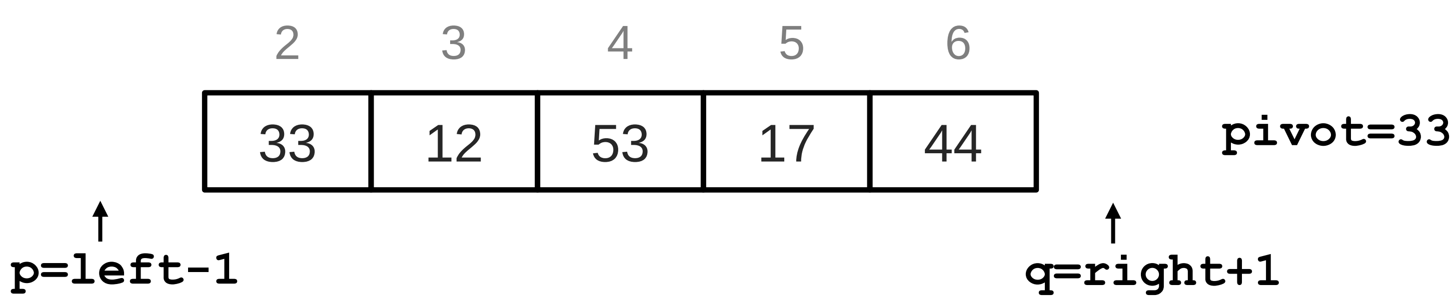
Partition Korrektheit (I)

zz. `A[left…q]=<pivot,A[q+1…right]>=pivot`
`q` Rückgabewert

Schleifeninvariante: Bei Eintritt in Zeile **`4`** der **`WHILE`**-Schleife enthalten **`A[left…p]`** nur Elemente ≤ **`pivot`** und **`A[q…right]`** nur ≥ **`pivot`**

vor den **`REPEAT`**s

Induktionsbasis:
Gilt vor erstem Eintritt für „leere“ Arrays mit **`p=left-1`** und **`q=right+1`**



Partition Korrektheit (II)

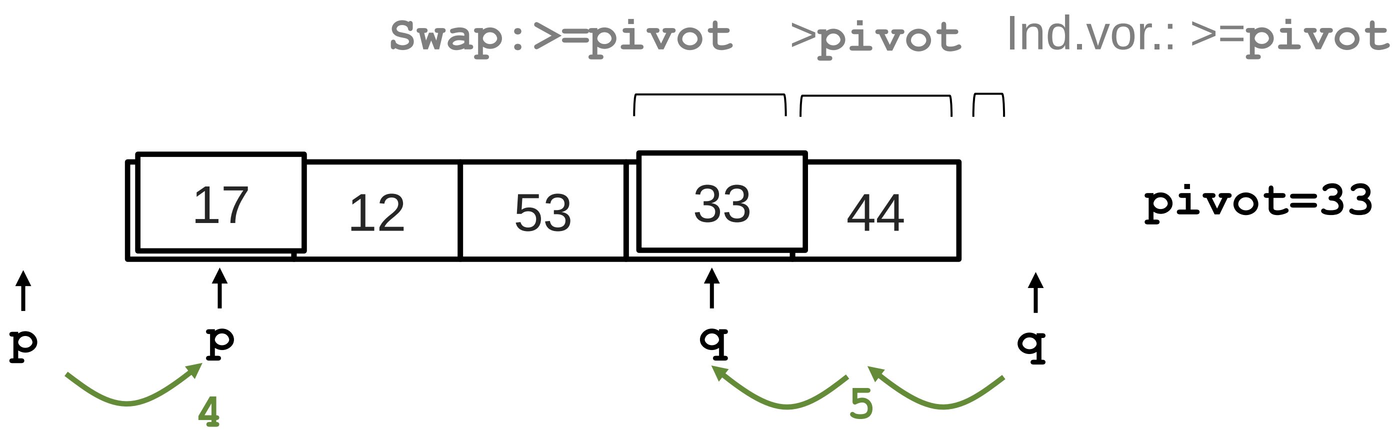
zz. **A[left…q]=<pivot,A[q+1…right]>=pivot**
q Rückgabewert

Schleifeninvariante: Bei Eintritt in Zeile **4** der **WHILE**-Schleife enthalten **A[left…p]** nur Elemente ≤ **pivot** und **A[q…right]** nur ≥ **pivot**

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

In voriger Iteration:
p lief in **4** nur über Werte <**pivot**, **q** in **5** nur über Werte >**pivot**,
bis schließlich **A[p]>=pivot** und **A[q]=<pivot**

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



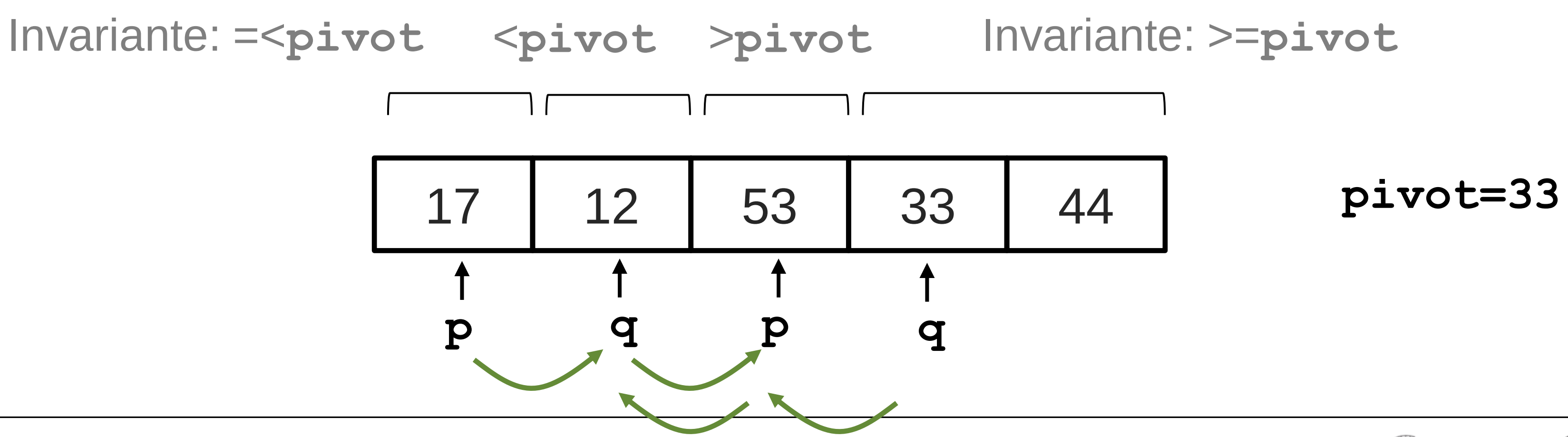
Partition Korrektheit (III)

zz. **A[left…q]=<pivot,A[q+1…right]>=pivot**
q Rückgabewert

Betrachte letzte Iteration von **WHILE**, in der **q=<p** wird

Bei Eintritt **A[left…p]=<pivot**, **A[q…right]>=pivot** wegen Invariante,
p läuft in **4** nur über Werte<**pivot**, **q** in **5** nur über Werte>**pivot**,
bis schließlich **A[p]>=pivot** und **A[q]=<pivot**

Dann **A[left…p-1]=<pivot, A[q+1…right]>=pivot** nach **REPEAT**s



Partition Korrektheit (III)

zz. `A[left…q]=<pivot,A[q+1…right]>=pivot`
`q` Rückgabewert

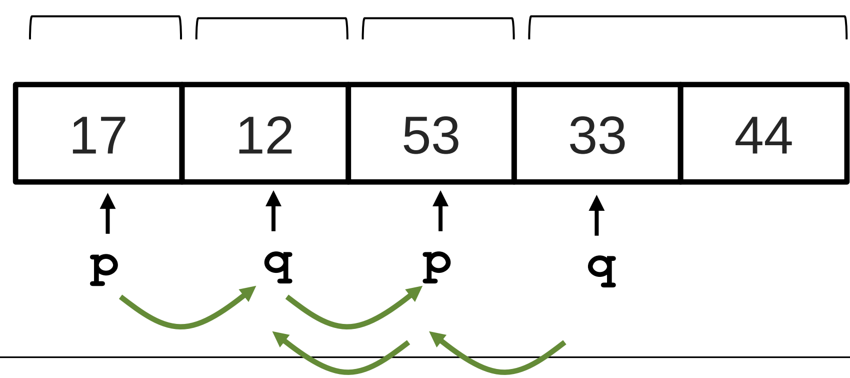
Betrachte letzte Iteration von **`WHILE`**, in der **`q=<p`** wird

Dann **`A[left…p-1]=<pivot`**, **`A[q+1…right]>=pivot`** nach **`REPEAT`**s

Da Abbruch von **`REPEAT`**, wenn **`>=pivot`** bzw. wenn **`=<pivot`**, entweder **`q=p`** (Beispiel #2) oder **`p=q+1`** (Beispiel #3/hier)

Im Fall **`p=q+1=p+1`** gilt also **`A[left…q]=A[left…p-1]=<pivot`** und **`A[p…right]=A[q+1…right]>=pivot`**

Invariante: =<**`pivot`** <**`pivot`** >**`pivot`** Invariante: >=**`pivot`**



`pivot=33`

`7 return q`

Partition Korrektheit (IV)

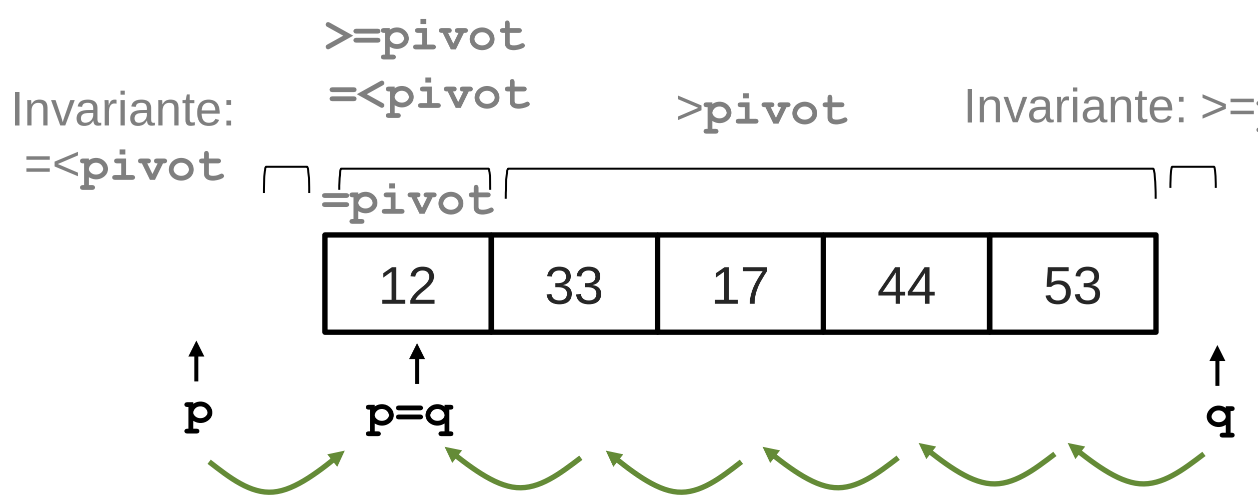
zz. **A[left…q]=<pivot,A[q+1…right]>=pivot**
q Rückgabewert

Betrachte letzte Iteration von **WHILE**, in der **q=<p** wird

Dann **A[left…p-1]=<pivot**, **A[q+1…right]>=pivot** nach **REPEAT**s

Da Abbruch von **REPEAT**, wenn **>=pivot** bzw. wenn **=<pivot**, entweder **q=p** (Beispiel #2/hier) oder **p=q+1** (voriges Beispiel)

Im Fall **q=p** gilt **A[p]=A[q]=pivot** und daher
A[left…p]=A[left…q]=<pivot, **A[p+1…right]=A[q+1…right]**
>=pivot



pivot=12

7 return q

Partition Korrektheit (V)

noch zz.: `left=<q<right`, nicht-triviale Aufteilung

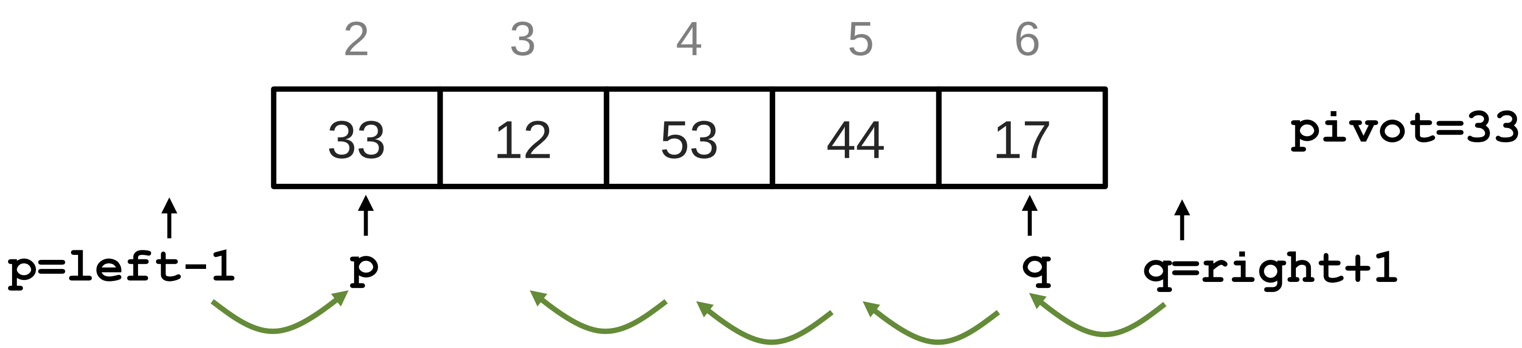
`p=q=right` kann passieren, wenn **`WHILE`**-Schleife nur einmal ausgeführt; sonst würde nächste Iteration von **`WHILE`** Wert `q` weiter erniedrigen

Andererseits `p=left` nach 1.Iteration von **`WHILE`**, da `A[left]=pivot`

Wegen `p=left<right` würde `q=right` weitere **`WHILE`**-Iteration bedeuten

nur solche Eingaben in `Partition` erlaubt

Also definitiv `q<right`

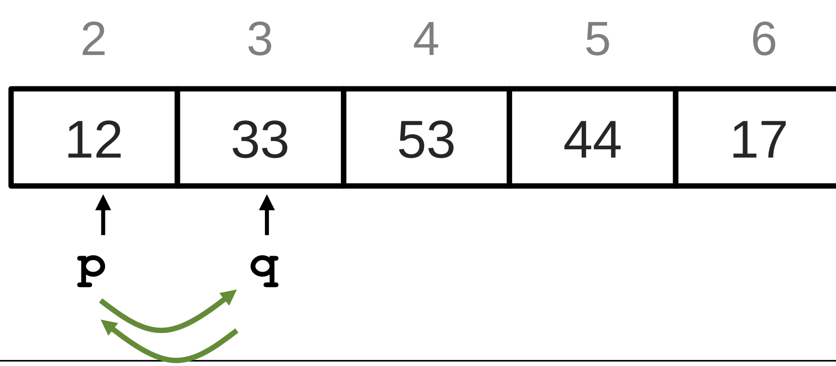


Partition Korrektheit (VI)

noch zz.: `left=<q<right`, nicht-triviale Aufteilung

Zusätzlich kann `q<left` nicht passieren, da

in allen `WHILE`-Iterationen (auch in letzter, in der `q=<p` wird)
vor `REPEAT` links von `q` immer Wert `=<pivot` steht (vgl. Terminierung),
`REPEAT` also nur bis `left` runterzählen kann



`pivot=33`

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 $\Omega(n)$

Laufzeit Partition $\Theta(n)$

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

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

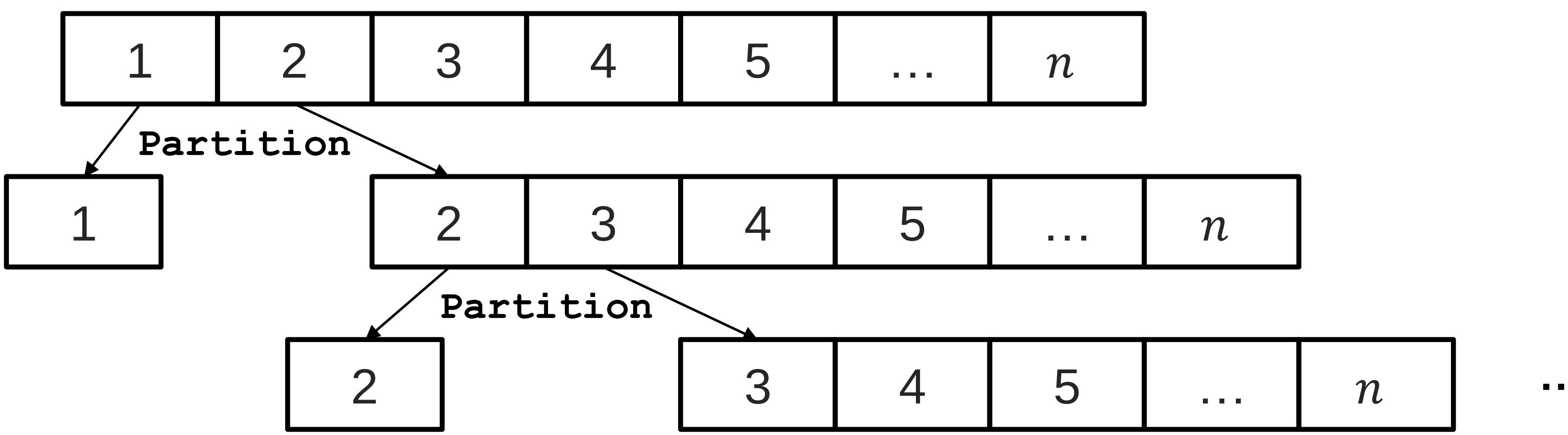
Laufzeitanalyse Quicksort: Worst-Case, Average-Case und erwartete Laufzeit

Laufzeit im Worst-Case: Untere Schranke

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

IF left<right THEN  //more than one el
  q=partition(A,left,right);   // q pa
  quicksort(A,left,q);         // sort
  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

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

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

$$T(n) \leq T(n-1) + dn$$

Aufwand für $T(1)$, **`IF`** und ggf. **`Partition`**

$$\leq T(n-2) + d(n-1) + dn$$

$$\vdots$$

$$\leq \sum_{i=1}^{n} di = O(n^2)$$

Laufzeit im Worst-Case: Obere Schranke (II)

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

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

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

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

i nicht-trivialer Partitionsindex (daher Induktion anwendbar)

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

Induktions-schritt:

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

maximal für $i = 1$

Best-Case für Quicksort (I)

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

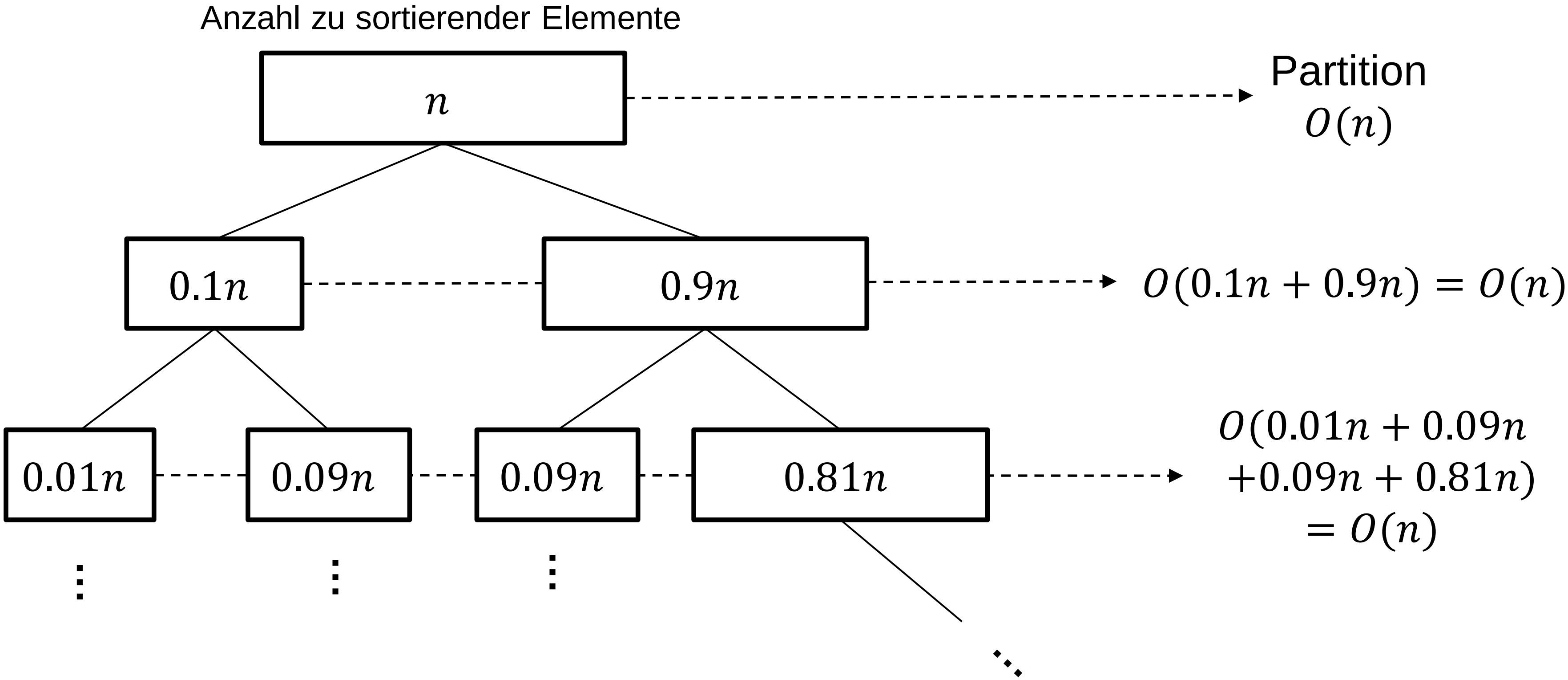
$$T(n) = 2T(n/2) + \Theta(n) \implies \text{Laufzeit } \Theta(n \log n)$$

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

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

Best-Case für Quicksort (II)

„Rekursionsbaum“



längster Pfad bis $n/0.9^k \leq 1$,
also $k = \lceil \log_{0.9} n \rceil = O(\log n)$

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

Average-Case-Laufzeiten?

Achtung: übliche Definition ist komplizierter

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

Intuitiver Ansatz:
$T(n)$ = $E_{D(n)}$ [#Schritte für x]

erwartete Anzahl von Schritten über Verteilung $D(n)$ auf Eingabedaten der Komplexität n

Wie verhält sich Quicksort im Durchschnitt auf „zufälliger“ Eingabe?

Für zufällige Permutation $D(n)$ eines fixen Arrays von n Elementen benötigt Quicksort $E_{D(n)}[\,Anzahl\ Schritte] = O(n \log n)$

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
j=RANDOM(left,right); Swap(A[left],A[j]);
                              //j uniform in [left…right]
pivot=A[left];
p=left-1; q=right+1;  //move from left resp. right
WHILE p<q DO
   REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
   REPEAT q=q-1 UNTIL A[q]=<pivot; //right down
   IF p<q THEN Swap(A[p],A[q]);
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 { #Schritte für x }

Erwartete Laufzeit
$T(n)$ = max { E_A[#Schritte für x] }

erwartete Anzahl von Schritten
(über zufällige Wahl des Algorithmus A)
für „schlechteste" Eingabe
der Komplexität n

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

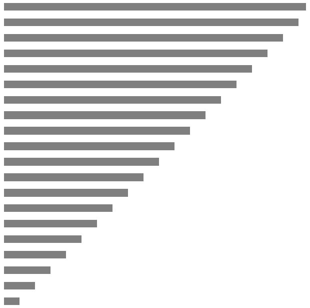
Intuition:
zufällige Wahl des Pivot-Elementes teilt Array im Durchschnitt mittig,
unabhängig davon, wie Array aussieht

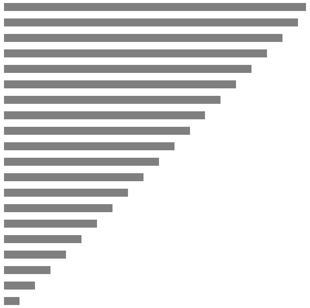
Merge Sort vs. Randomized-Quicksort (I)

Merge Sort | Randomized-Quicksort

„unsortierte Eingabe“

„schlechte Eingabe“





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

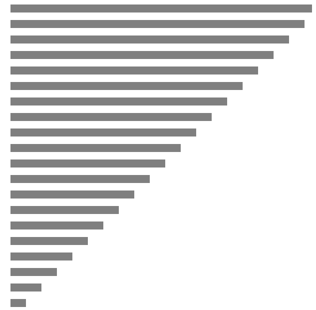
Merge Sort vs. Randomized-Quicksort (II)

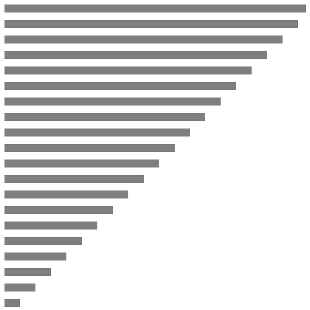
Merge Sort

Randomized-Quicksort

„unsortierte Eingabe“

„schlechte Eingabe“





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 $\Theta(n \log n)$	Wort-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`**

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

done=false;
WHILE !done DO
    determine (i,j); // arbitrarily
    comp(i,j); // returns A[i] =< A[j]?
    set done; // true or false
compute I from comp-information only;
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

done=false;
WHILE !done DO
    determine (i,j); // arbitrarily
    comp(i,j); // returns A[i] =< A[j]?
    set done; // true or false
compute I from comp-information only;
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, π**`(A)`**, für alle π:

π(0)	π(1)	...						π(n-1)

Insgesamt also $n!$ viele mögliche Eingabe-Arrays

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

Mögliche Ausgaben

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

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

Annahme:
macht m Vergleiche

es gibt maximal
2^m mögliche
Antwortsequenzen
Ja/Nein

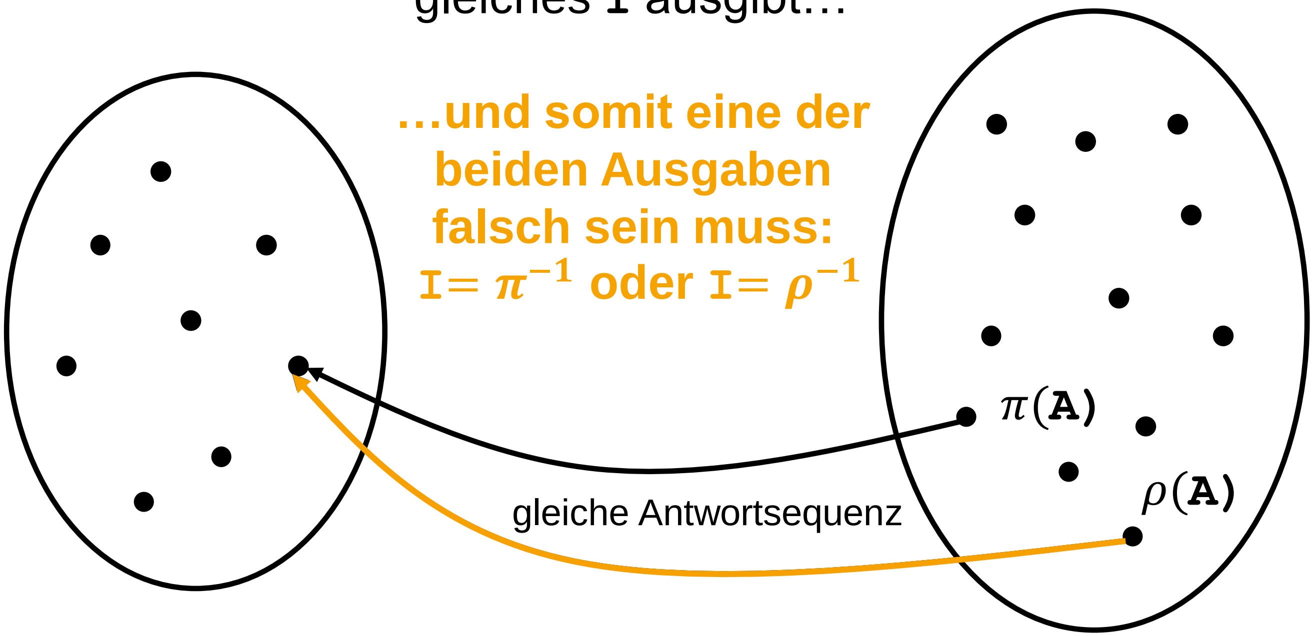
einzige Info
über **A**
(außer n)

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

Algorithmus gibt
maximal 2^m verschiedene
Ausgaben **I**

Ein- und Ausgabeverhältnis

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



Algorithmus gibt maximal 2^m verschiedene Ausgaben `I`

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

Schranke ausrechnen

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

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

Stirling-Approximation:
$n! \geq \left(\frac{n}{e}\right)^n$

bzw. $m = \Omega\left(n \cdot \log(n)\right)$

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?

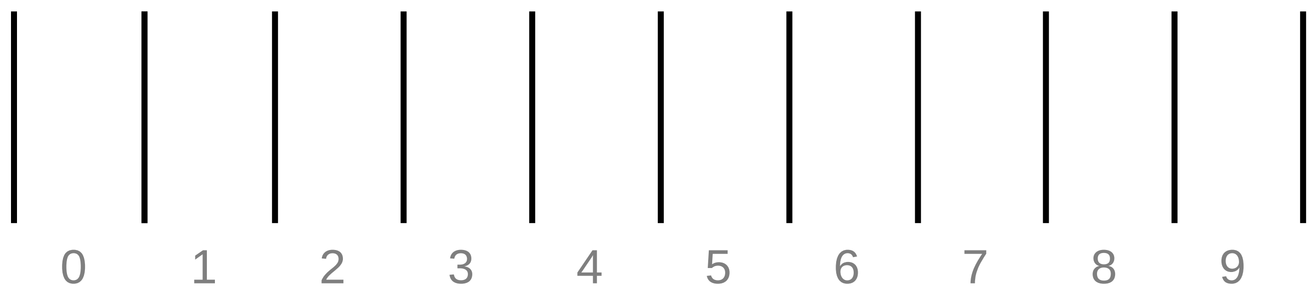
Radix-Sort:
Sortieren in linearer Zeit (?)

Ansatz

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

543	123	017	445	333	225	711	400	876

d=3
D=10



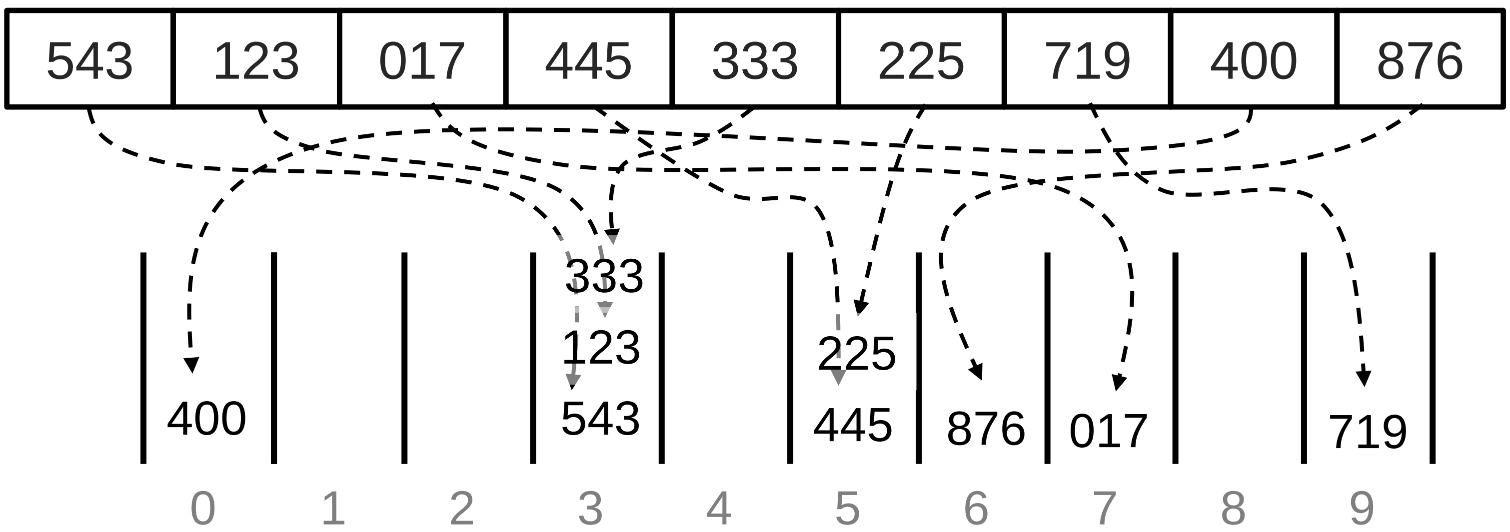
D
„buckets“

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

Queues
(FIFO-Systeme)
→ Abschnitt 3

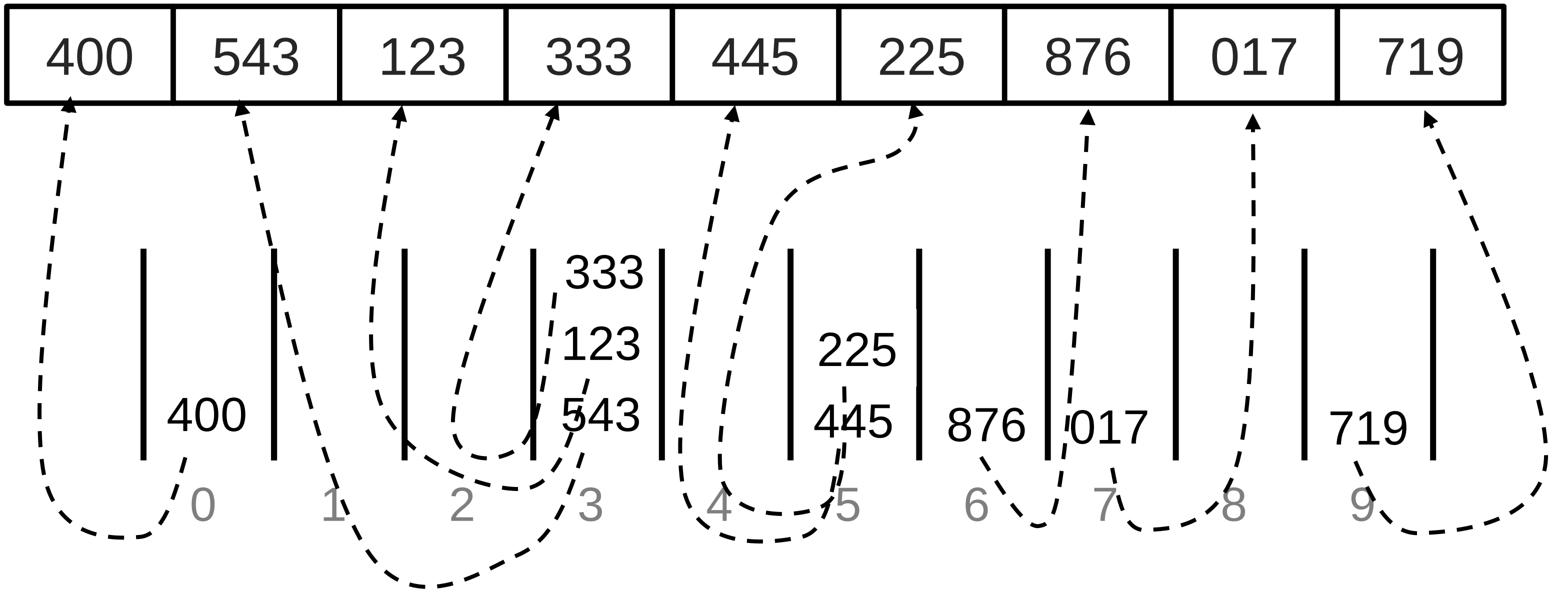
Erste Iteration (I)

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



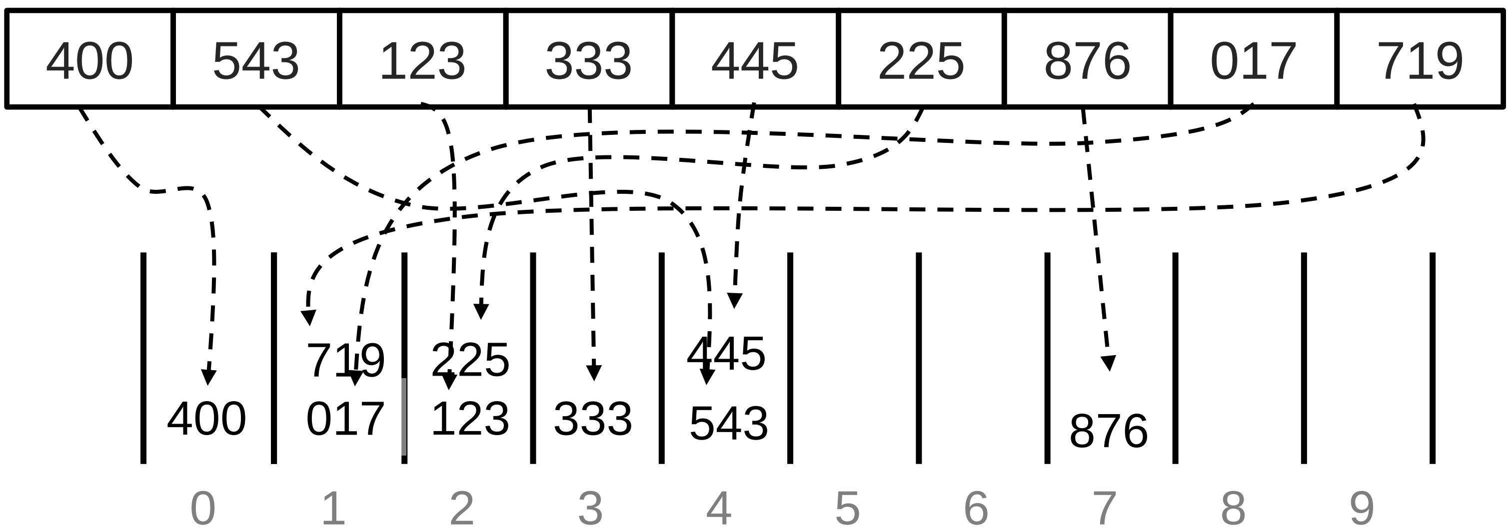
Erste Iteration (II)

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



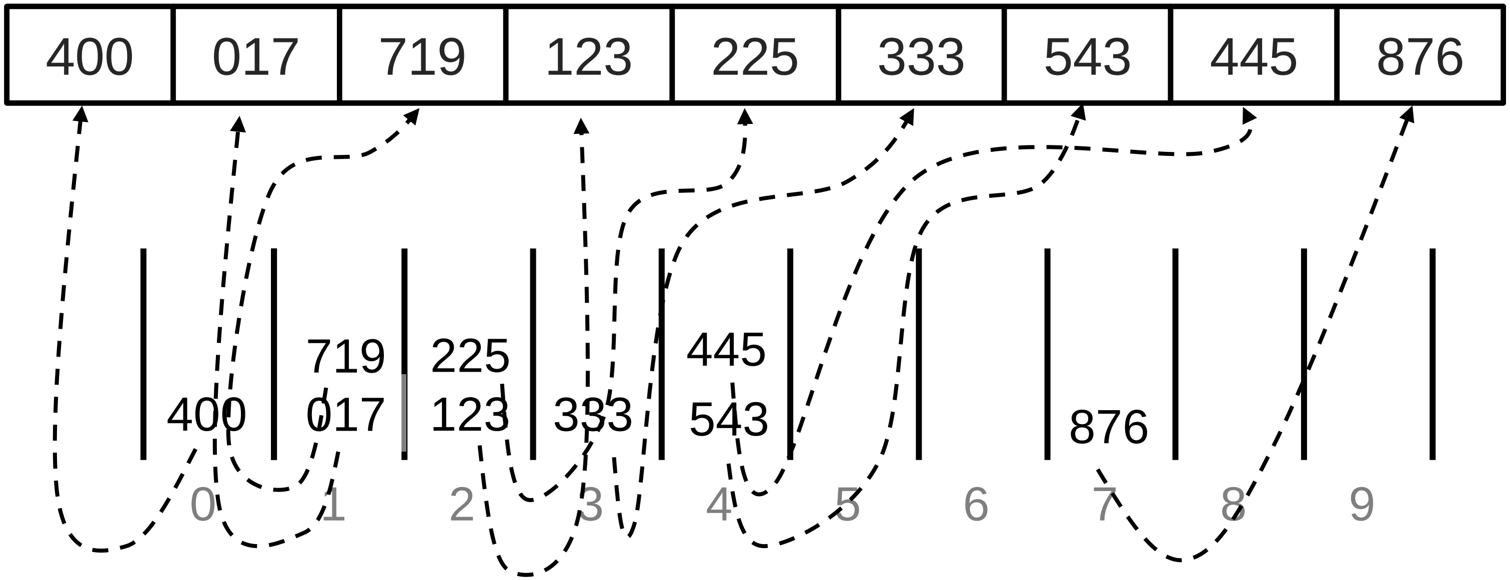
Zweite Iteration (I)

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



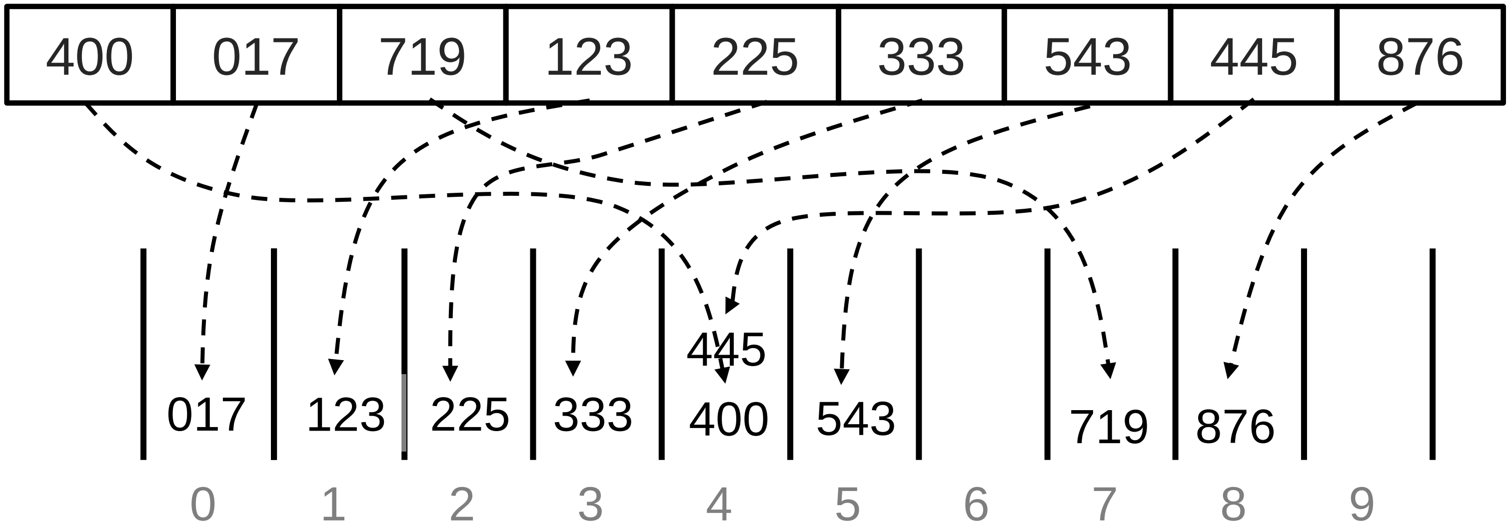
Zweite Iteration (II)

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



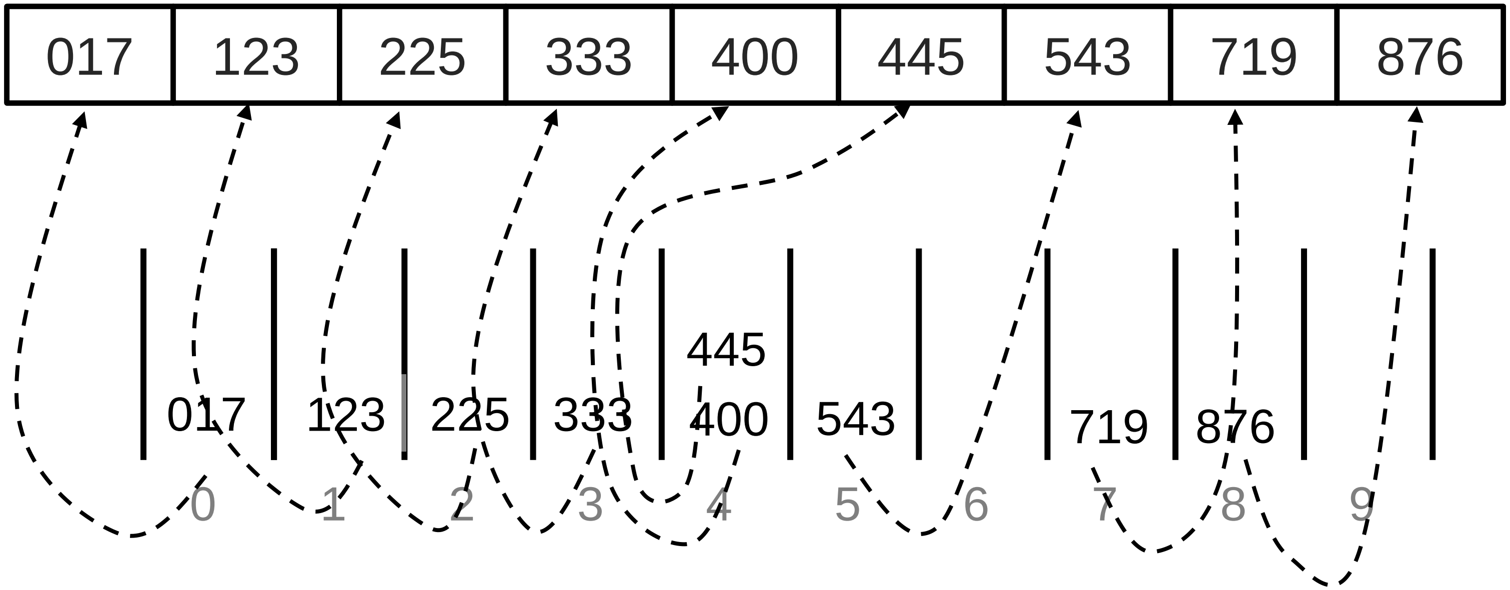
Dritte Iteration (I)

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)

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

`A.size=`Anzahl eingetragender Elemente in Array **`A`**

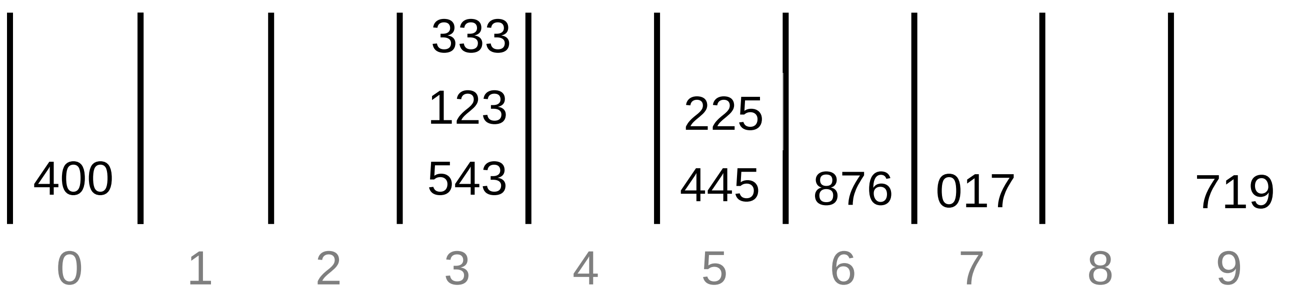
```
putBucket(A,B,i,j) // call-by-reference
z=A[j].digit[i]; // i-th digit of A[j]
b=B[z].size;     // next free spot
B[z][b]=A[j];
B[z].size=B[z].size+1;
```

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

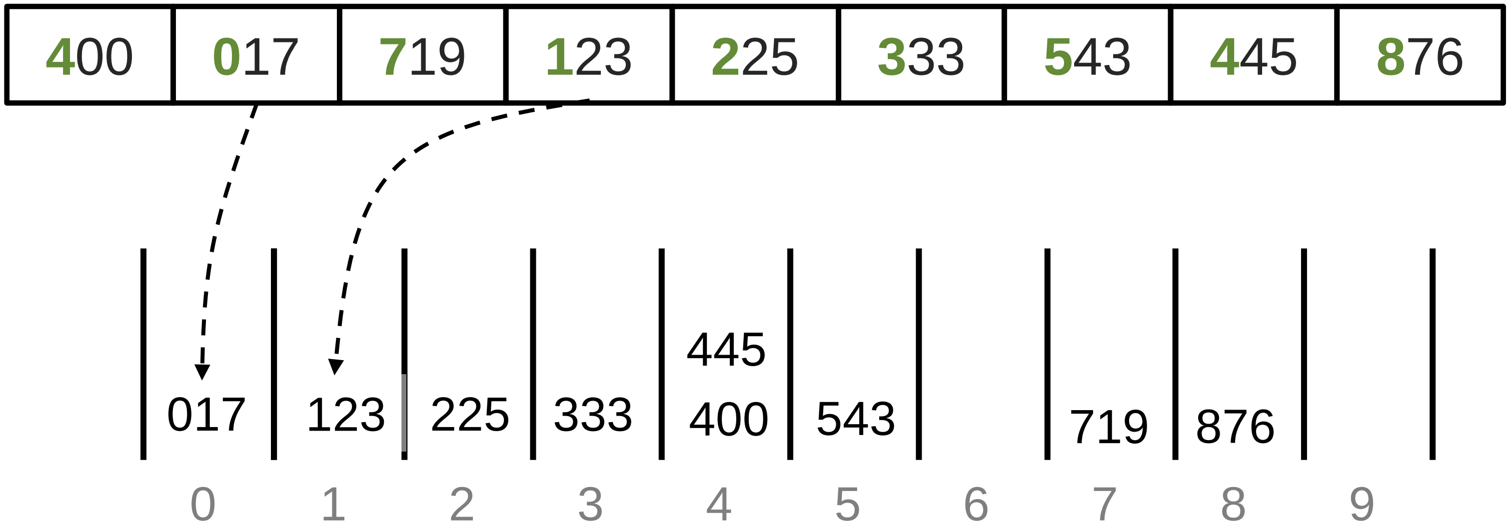
40**0**	54**3**	12**3**	33**3**	44**5**	22**5**	87**6**	01**7**	71**9**



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

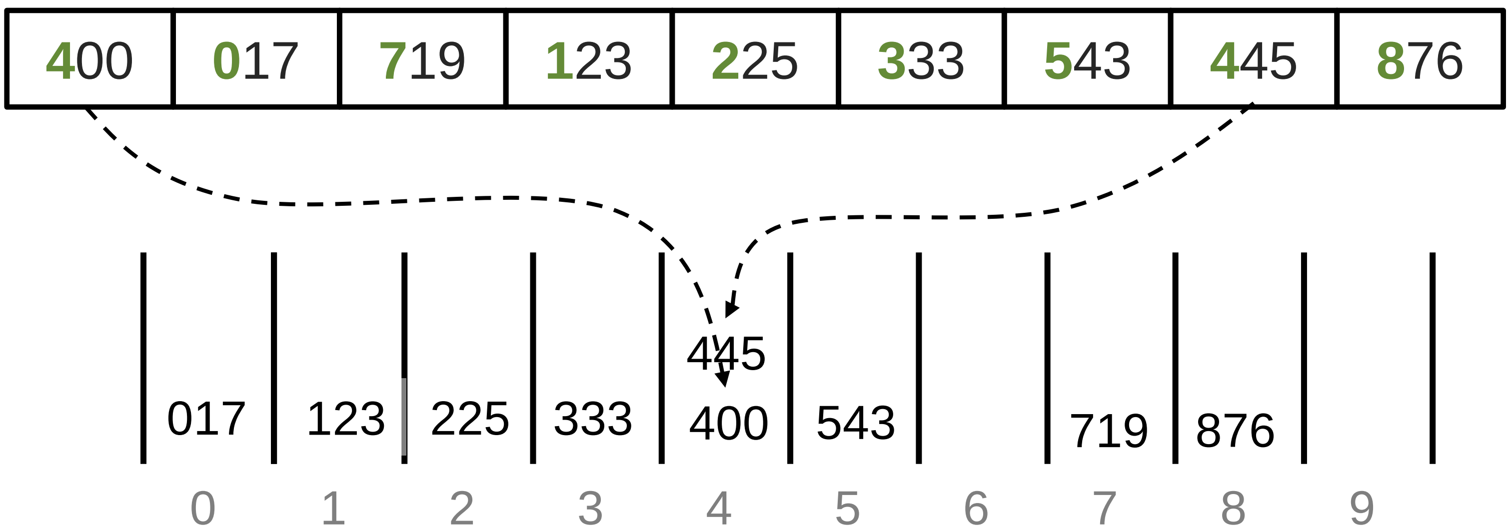


Induktionsschritt i →i+1 (1.Fall):
Wenn (i+1)-te Ziffer zweier beliebiger Werte verschieden,
dann wird Wert mit kleinerer (i+1)-ten Ziffer weiter vorne einsortiert,
zuerst wieder ausgelesen und steht somit auch im Array vorher

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 →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

Gesamtlaufzeit
$O(d \cdot (n + D))$

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

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

$O(n)$

$O(n)$ Schritte (alles in **A** kopieren)

$O(D)$

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

O(1)

Laufzeit – Interpretation

Gesamtlaufzeit
$O(d \cdot (n + D))$

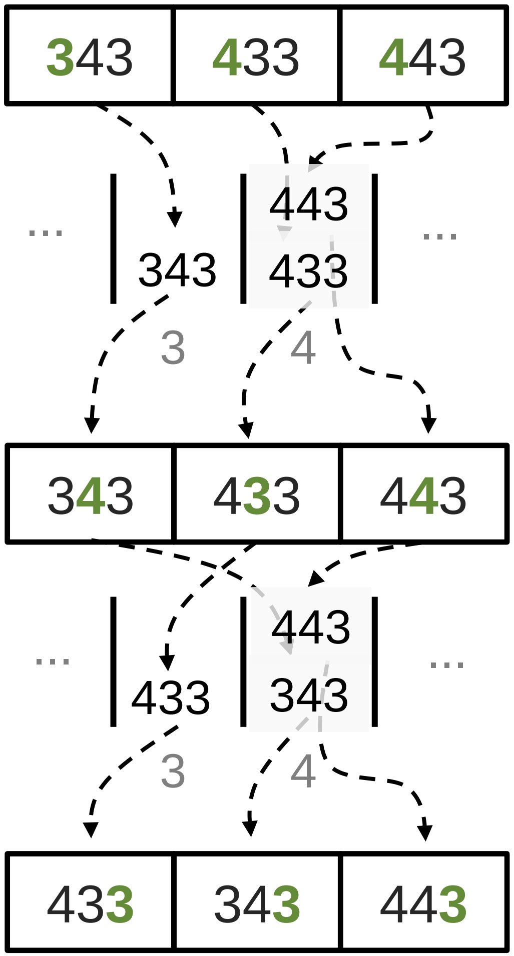
Größe Zifferbereich D oft als konstant angesehen: Laufzeit $O(dn)$

Wenn auch d als konstant angesehen: Laufzeit $O(n)$

linear!

Aber:
eindeutige Schlüssel für n Elemente
benötigen $d = \Theta(\log_D n)$ Ziffern! Laufzeit $O(n \cdot \log n)$

Mit höchstwertiger Ziffer beginnen?



…folgende Iteration ändert Reihenfolge nicht mehr

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

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