Algorithmen und Datenstrukturen

TECHNISCHE UNIVERSITÄT DARMSTADT

Stefan Roth, SS 2025

02

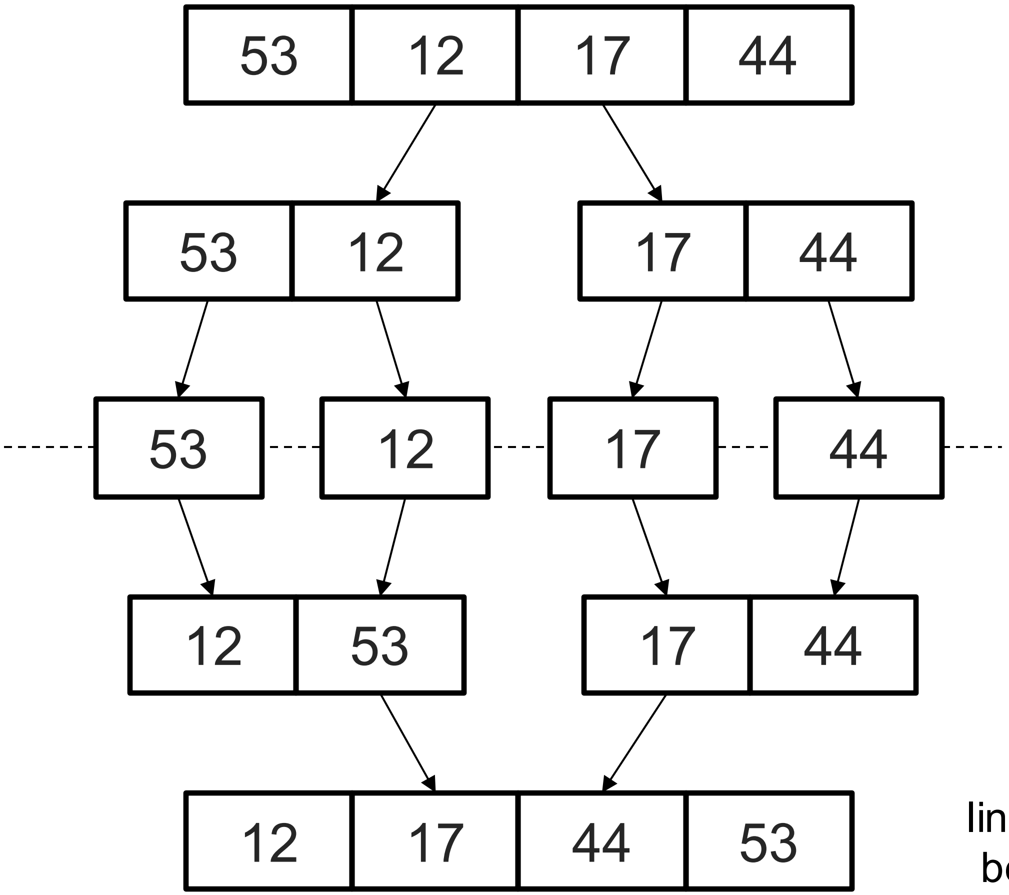
Sortieren

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

Merge Sort

Idee Divide & Conquer (& Combine)

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

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
```

Laufzeitanalyse: Rekursionsgleichungen

Laufzeitabschätzung Merge Sort

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

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

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

zwei rekursive Aufrufe für jeweils halb so großes Array

(ignorieren hier zur Vereinfachung Runden)

IF + **floor**
(konstanter Aufwand)

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

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

TECHNISCHE UNIVERSITÄT DARMSTADT

vi SYSTEMS

Rekursion „manuell iterieren“

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

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

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

Allgemeiner Ansatz: Mastermethode

Allgemeine Form der Rekursionsgleichung:

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

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

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

Interpretation:

Problem wird in a Teilprobleme der Größe n/b aufgeteilt

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

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

Mastertheorem

nach Cormen et al., Introduction to Algorithms

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

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

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

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

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

$a = 4, b = 3,\ \log_b a = 1.26 \ldots, \varepsilon = 0.26 \ldots$

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

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

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

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

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

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

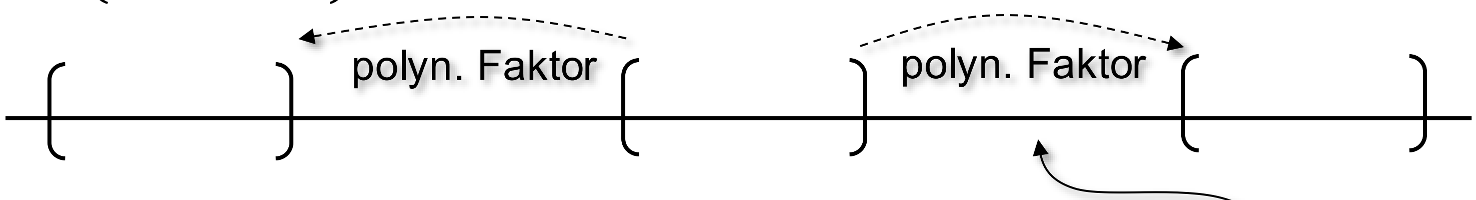
$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

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

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



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

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

---

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

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

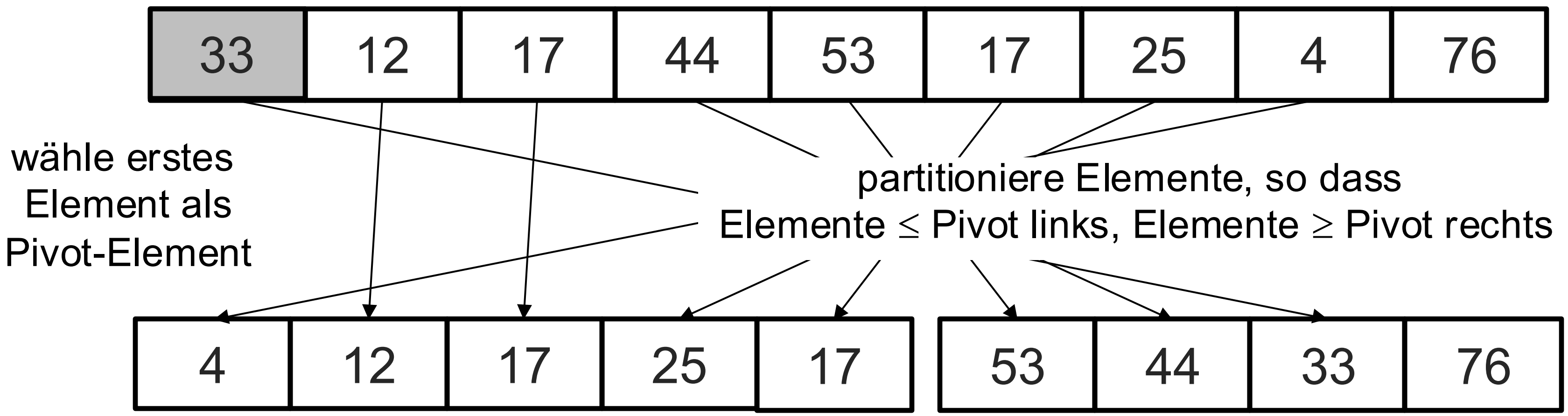
mit Hilfe des Mastertheorems.

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

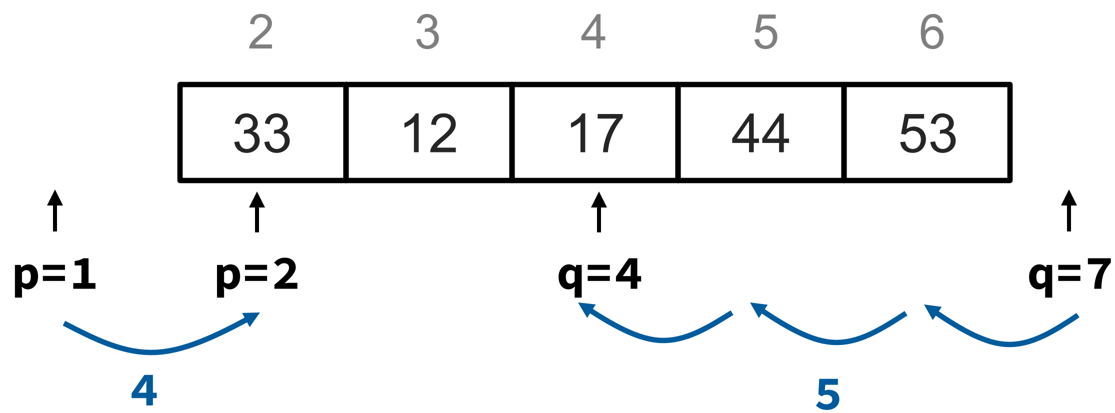
Algorithmus: Quicksort

```
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)

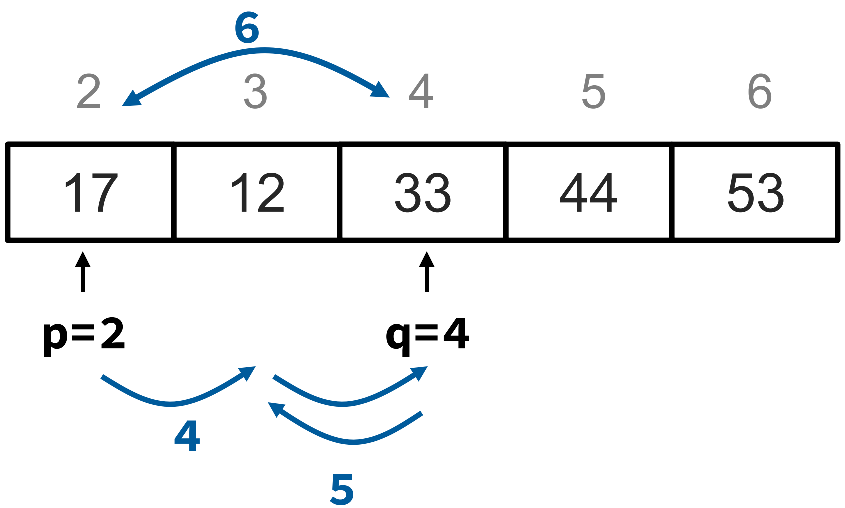
```
pivot=33
left=2, right=6
```




```
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

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
```

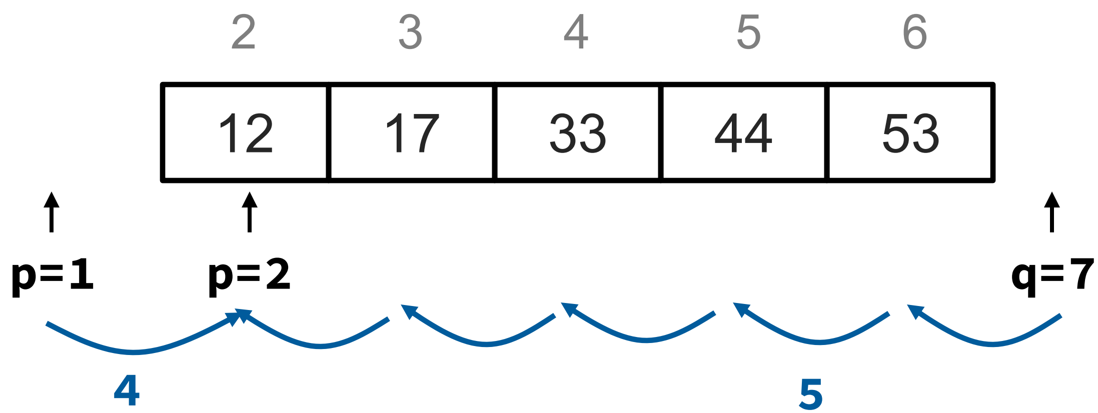
2	3	4	5	6
17	12	33	44	53
	q=3	p=4		

```
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

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 #2: Partition (II)

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

2	3	4	5	6
12	17	33	44	53

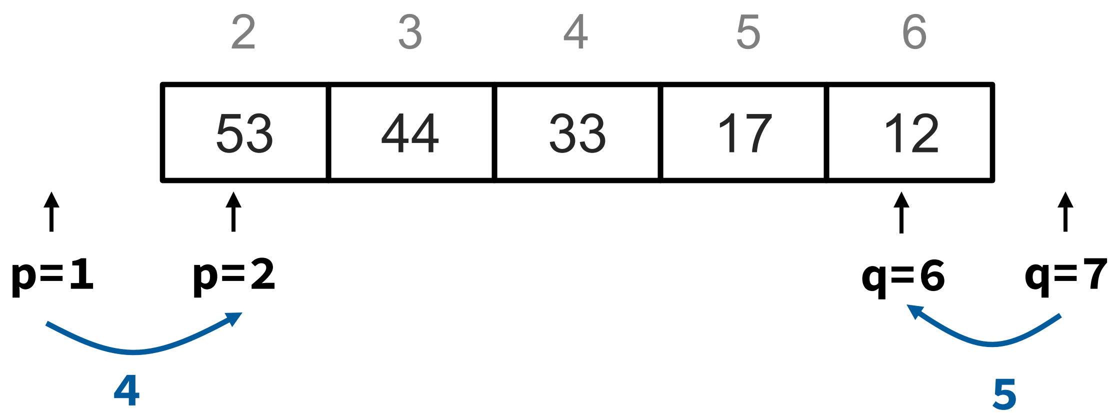
```
p=q=2
```

```
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)

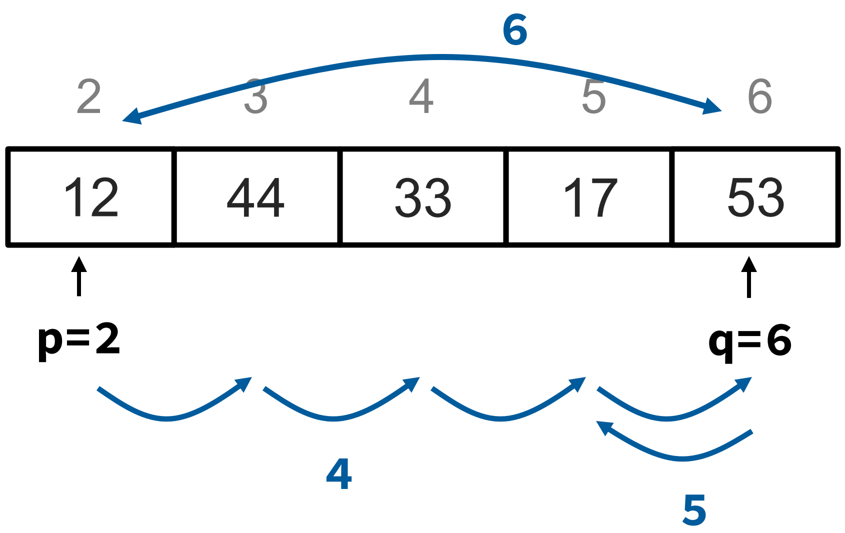
```
pivot=53
left=2, right=6
```




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

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
```

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**

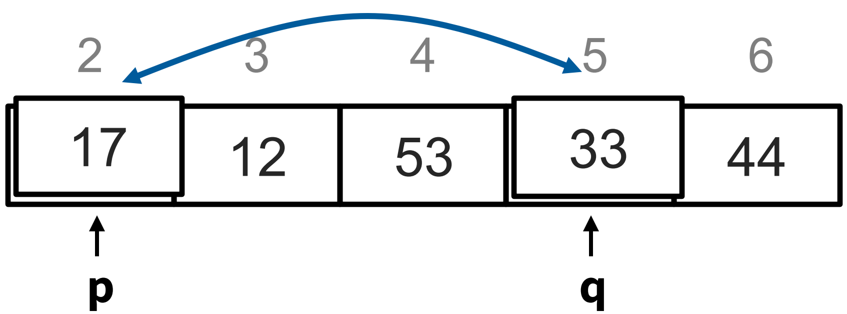
Behauptung: Vor Eintritt in **4** steht rechts von **p** stets Element ≥ **pivot**

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`**

2	3	4	5	6
33	12	53	17	44

`pivot=33`

↑ **`p=left-1`**

↑ **`q=right+1`**

Partition Korrektheit (II)

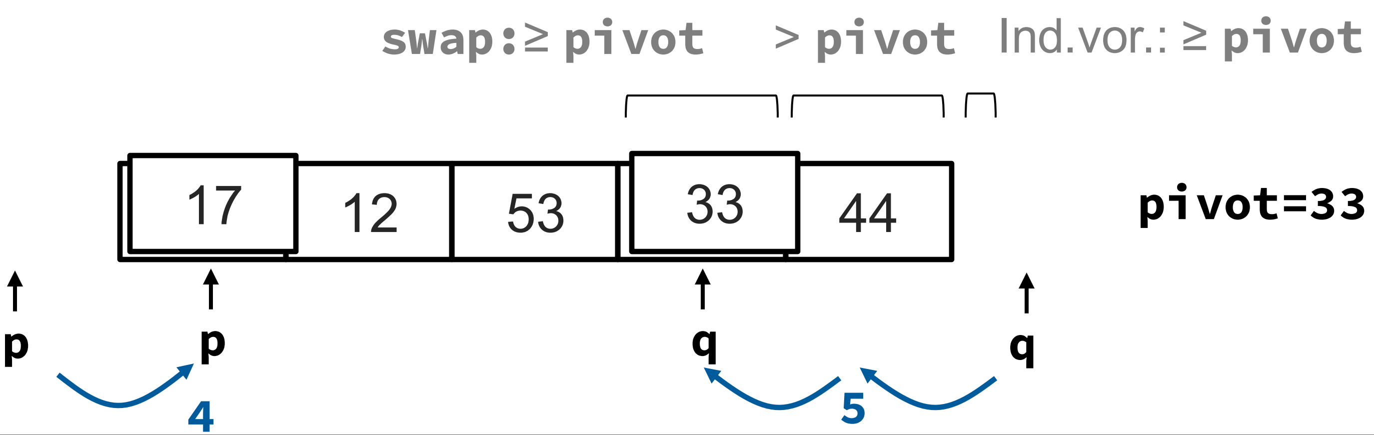
zz. `A[left…q]<=pivot, A[q+1…right]>=pivot`
`q` Rückgabewert

> 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



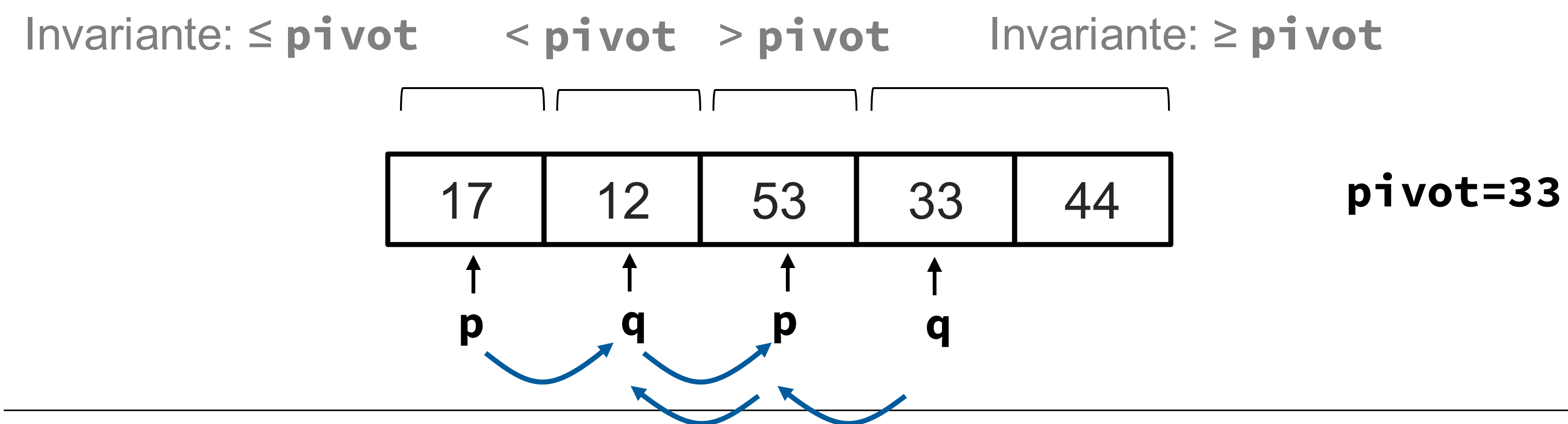
Partition Korrektheit (III)

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 **REPEATs**



Partition Korrektheit (III)

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

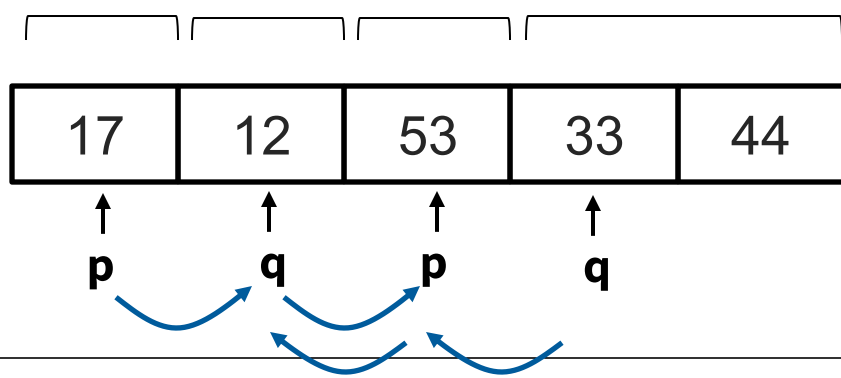
Betrachte letzte Iteration von **WHILE**, in der **q<=p** wird

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

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

Im Fall **p=q+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)

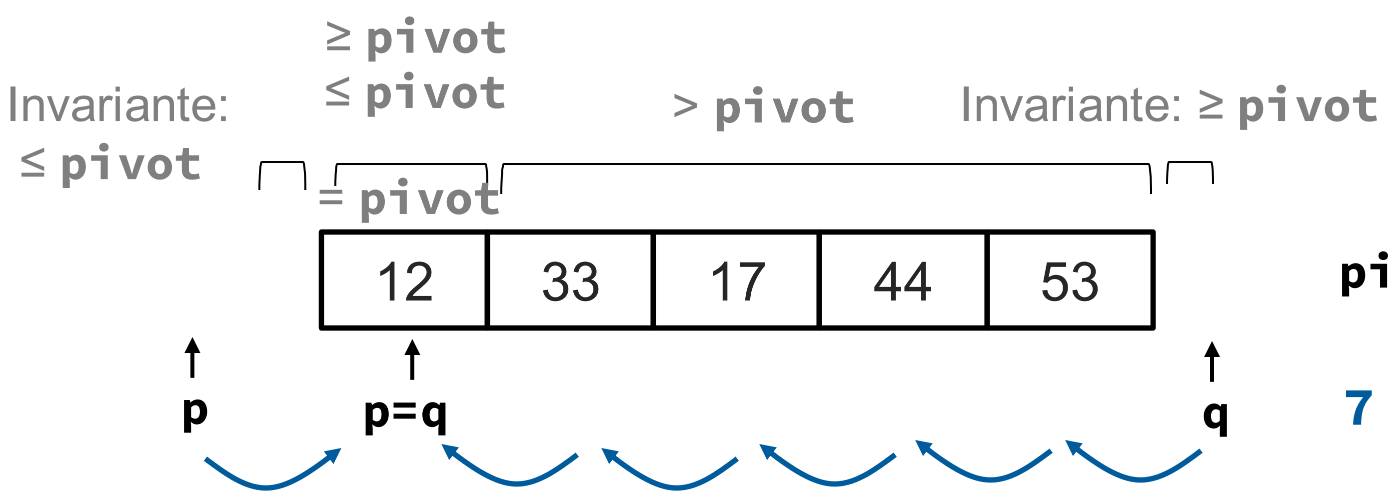
zz. `A[left…q]<=pivot, A[q+1…right]>=pivot`
`q` Rückgabewert

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 **`A[p]>=pivot`** bzw. wenn **`A[q]<=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

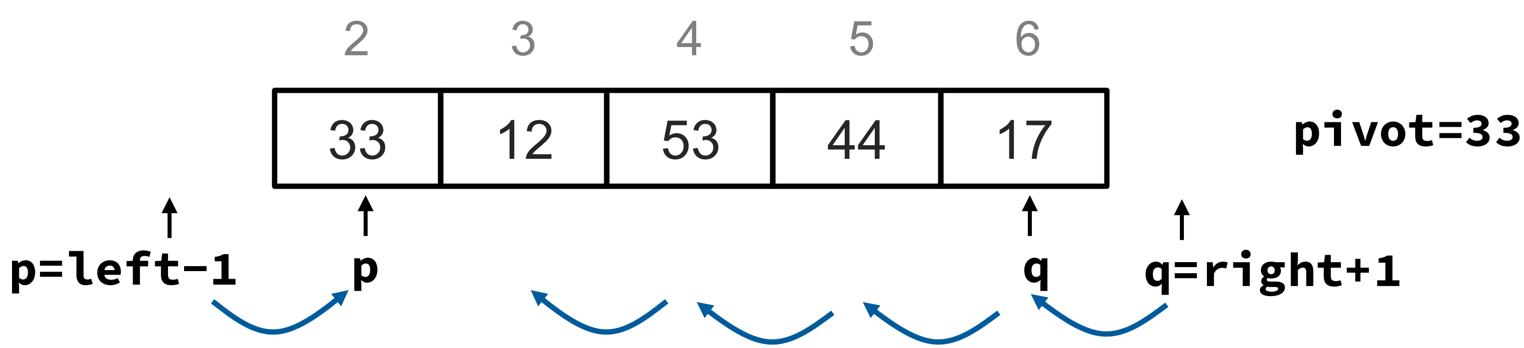
`p=q=right` kann nur passieren, wenn **`WHILE`**-Schleife nur einmal ausgeführt; sonst würde nächste Iteration von **`WHILE`** Wert **`q`** weiter erniedrigen

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

2	3	4	5	6
12	33	53	44	17
↑ p	↑ q			

`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]
```

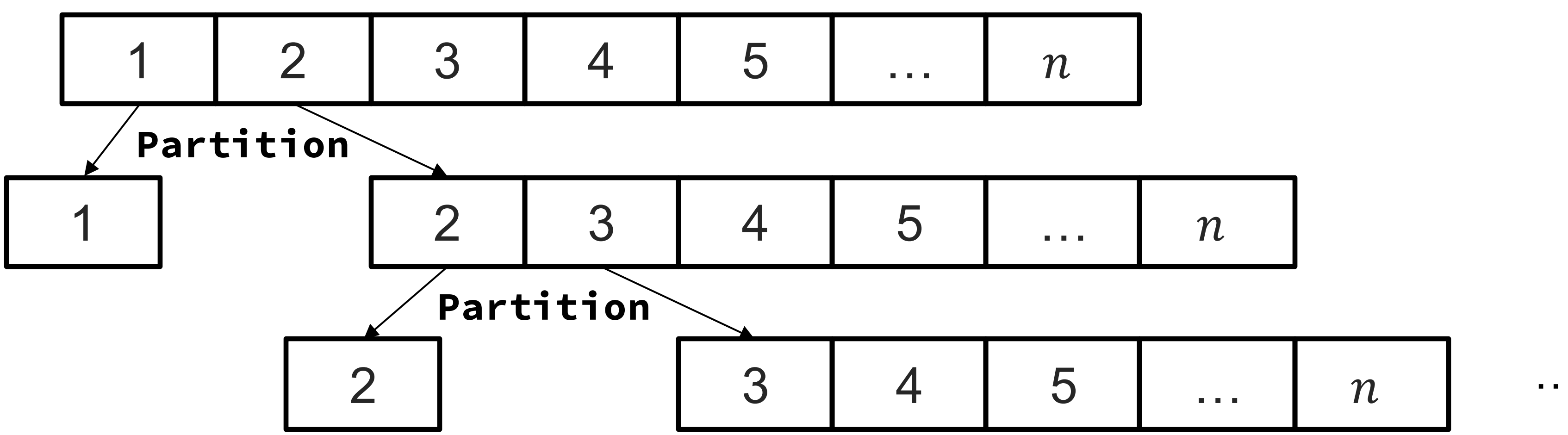
Laufzeitanalyse Quicksort: Worst-Case, Average-Case und erwartete Laufzeit

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)



TECHNISCHE UNIVERSITÄT DARMSTADT

SYSTEMS

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

$$
\begin{aligned}
T(n) &\le T(n-1) + dn \\
&\le T(n-2) + d(n-1) + dn \\
&\;\vdots \\
&\le \sum_{i=1}^{n} di = \mathcal{O}(n^2)
\end{aligned}
$$

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

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 $T(n) \leq dn^2$

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

Induktions-schritt:

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

i nicht-trivialer Partitionsindex (daher Induktion anwendbar)

maximal für $i = 1$

Best-Case für Quicksort (I)

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

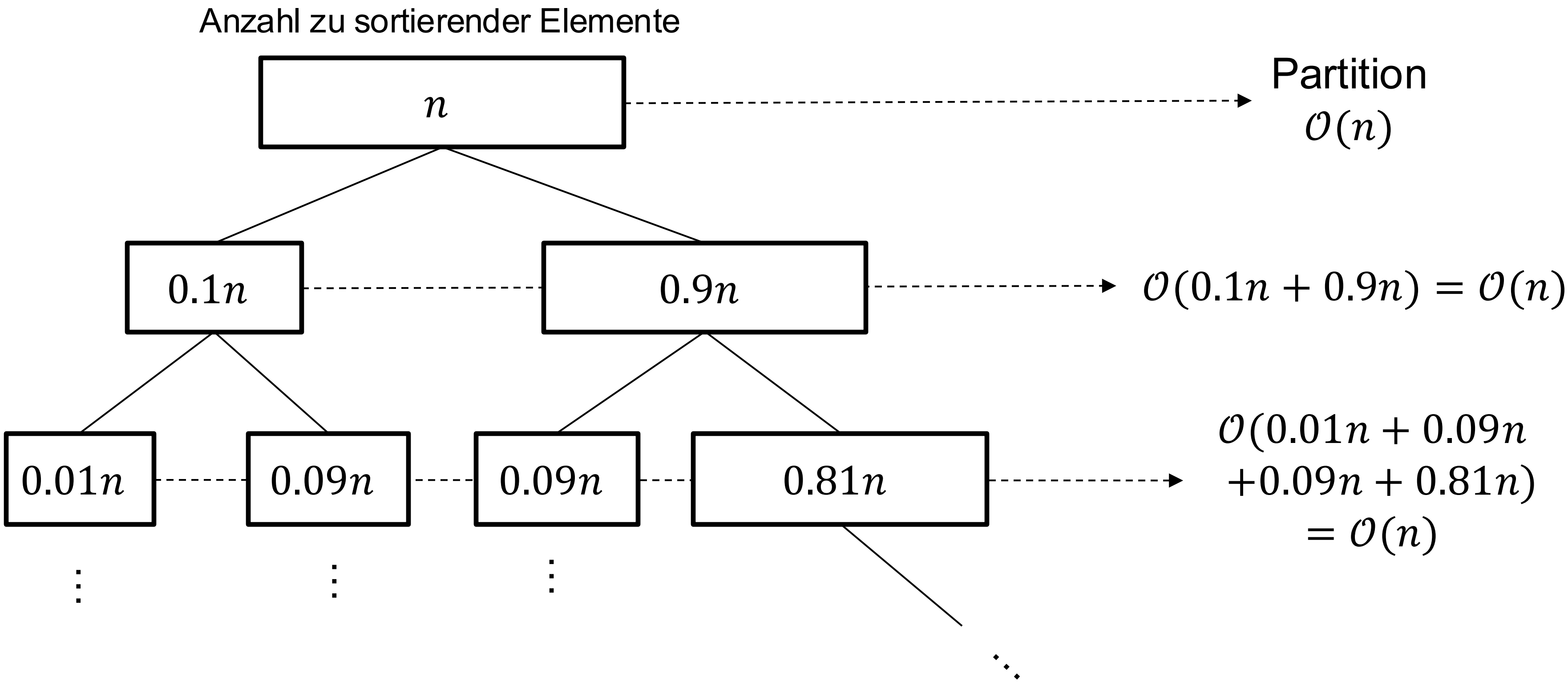
$$T(n) = 2T(n/2) + \Theta(n) \implies \text{Laufzeit } \Theta(n \log n)$$

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 $k \leq \lceil \log_{1/0.9} n \rceil = \mathcal{O}(\log n)$

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

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 $E_{D(n)}[\,Anzahl\ Schritte] = \mathcal{O}(n \log n)$

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
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 }

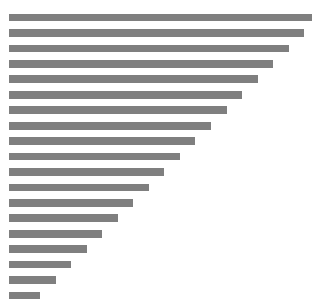
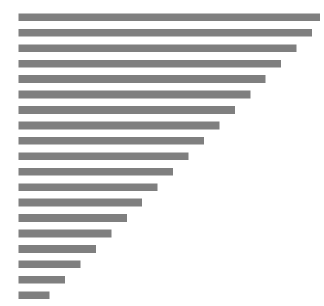
Erwartete Laufzeit
$T(n)$ = max { E_A[#Schritte für x] }

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

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

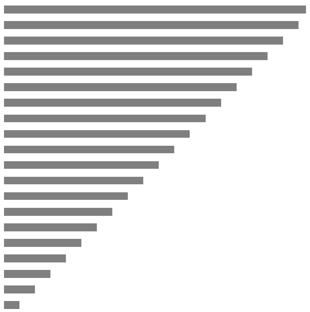
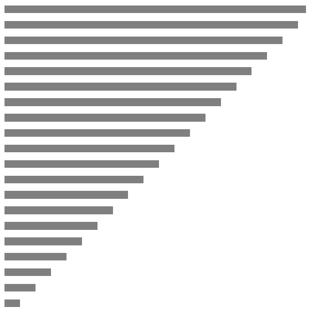
Intuition:
zufällige Wahl des Pivot-Elementes teilt Array im Durchschnitt mittig,
unabhängig davon, wie Array aussieht

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)

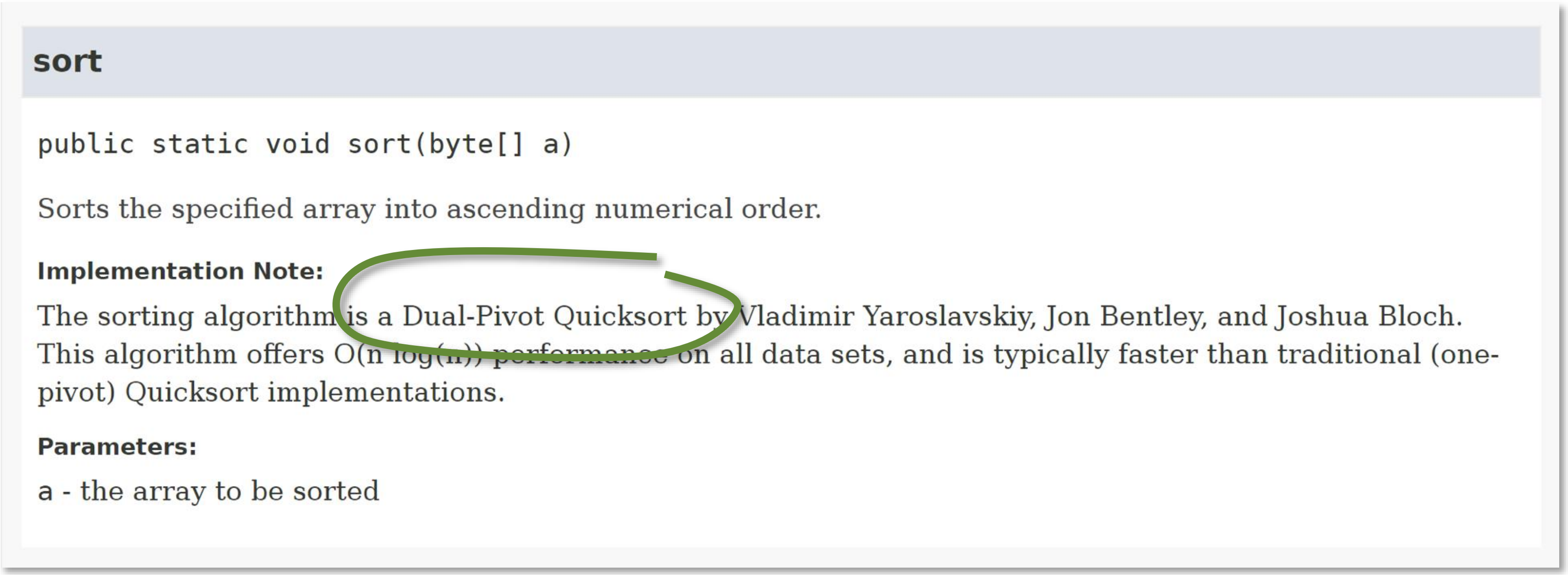
	Merge Sort	Randomized-Quicksort
„unsortierte Eingabe“		
„schlechte Eingabe“		

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



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:

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

Insgesamt also $n!$ viele mögliche Eingabe-Arrays

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

TECHNISCHE UNIVERSITÄT DARMSTADT

SYSTEMS

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)

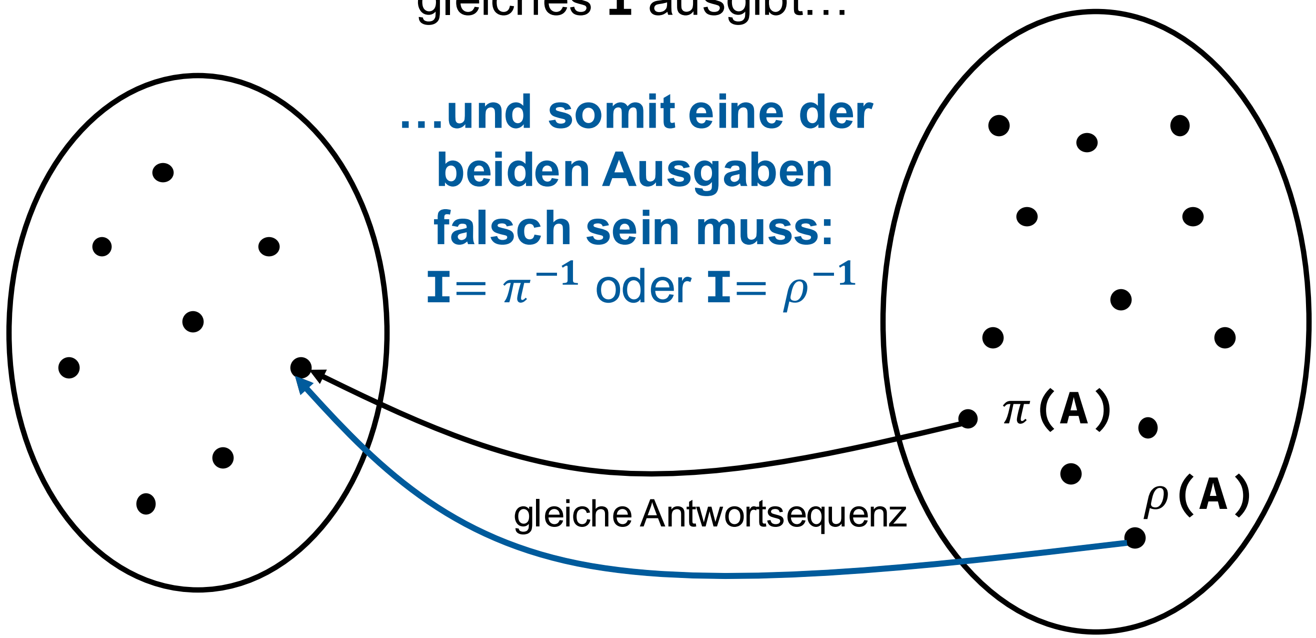
Determiniertheit:
gleiche Antwortsequenzen
(selbst für verschiedene Arrays)
führen zu gleicher Ausgabe

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

Ein- und Ausgabeverhältnis

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

…und somit eine der beiden Ausgaben falsch sein muss:
I$= \pi^{-1}$ oder **I**$= \rho^{-1}$



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

Es gibt $n!$ verschiedene Eingabe-Arrays π**(A)**

Schranke ausrechnen

Wenn $2^m < n!$, dann gibt es verschiedene Arrays π`(A)`und ρ `(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. $m = \Omega\,(n \cdot \log(n))$

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?