LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME UND DATA MINING

Kapitel 3: Sortieren

Einfache Sortierverfahren
Höhere Sortierverfahren
Problemspezifische Komplexitätsschranken

DBS Prof. Dr. Thomas Seidl

Motivation zum Sortieren

- Anwendungsbeispiele:
  - Priorisierung von Aufgaben zum Zeitmanagement
  - Packaufgaben, schwere/große Gegenstände zuerst einpacken
  - Statistische Übersicht

- Ausgangpunkt: Folge von *Datensätzen*

$$D_1, D_2, D_3, \dots, D_n$$

- Jeder Datensatz besitzt eine Schlüsselkomponente $D_i.key$
- Jeder Datensatz kann außerdem weitere Informationseinheiten enthalten (z.B. Name, Adresse, PLZ, etc.)
- Ziel: Datensätze derart anordnen, dass die Schlüsselwerte sukzessive ansteigen (oder absteigen)
- Bedingung: Schlüssel müssen vergleichbar sein!

Totale Ordnung

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

Das Paar $(M, \leq)$ heißt genau dann eine *totale (bzw. lineare) Ordnung* auf M, wenn folgende Eigenschaften erfüllt sind:

- Reflexivität: $\forall x \in M$: $x \leq x$
- Transitivität: $\forall x, y, z \in M$: $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Antisymmetrie: $\forall x, y \in M$: $x \leq y \wedge y \leq x \Rightarrow x = y$
- Totalität: $\forall x, y \in M$: $x \leq y \vee y \leq x$

Totale Ordnung: Beispiel

- $\leq$ auf den natürlichen Zahlen bildet eine totale Ordnung.

- Die lexikographische Ordnung $\leq_{lex}$ ist eine totale Ordnung:
  Sei $\leq$ die totale Ordnung auf den Buchstaben $A \leq B \leq \cdots \leq Z$.
  - Für zwei endliche Wörter $u = a_1 \dots a_k u_1 \dots u_m$ und $v = a_1 \dots a_k v_1 \dots v_n$ mit $u, v \in \{A, B, \dots, Z\}^*$ und $u_1 \neq v_1$ gilt:
    $$u \leq_{lex} v \Leftrightarrow u_1 \leq v_1 \vee u \text{ "leer"}$$
  - Beispiel: FRANZ $\leq_{lex}$ FRANZISKA $\leq_{lex}$ PAUL $\leq_{lex}$ PETER

- Die Teilmengenrelation $\subseteq$ auf Potenzmenge von $\mathbb{N}$ ist keine totale Ordnung.
  - Beweis: $\{1\} \not\subseteq \{2\}$ und $\{2\} \not\subseteq \{1\}$ verletzen Totalität.

Das Sortierproblem

Gegeben sei eine Folge von *Schlüsselwerten*

$$a_1, a_2, a_3, \dots, a_n$$

und eine totale Ordnung $\leq$ auf der Schlüsselmenge $\{a_1, \dots, a_n\}$.

Die Ausgabe des Sortierproblems ist dann die Folge der Schlüssel

$$a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \dots, a_{\pi(n)}$$

mit:

- $a_{\pi(i)} \leq a_{\pi(i+1)}$ für alle $1 \leq i \leq n$.
- π ist eine Permutation der Indexmenge $\{1, \dots, n\}$.

Das Sortierproblem: Beispiele

Quelle	Schlüsselelement	Ordnung
Telefonbuch	Nachname	Lexikographische Ordnung
Klausurergebnisse	Punktezahl	$\leq$ auf rationalen Zahlen
Lexikon	Stichwort	Lexikographische Ordnung
Studierendenverzeichnis	Matrikelnummer	$\leq$ auf $\mathbb{N}$
Entfernungstabelle	Distanz	$\leq$ auf $\mathbb{R}$
Fahrplan	Abfahrtszeit	„früher als“

Vergleichskriterien für Sortieralgorithmen

Sortieralgorithmen können nach verschiedenen Kriterien klassifiziert werden:

- Berechnungsaufwand
  - $O(n^2)$, $O(n \log n)$, $O(n)$
- Worst Case vs. Average Case
  - Ausnutzen von Vorsortierungen
- Speicherbedarf
  - In-place / in situ
  - Kopieren
- Stabilität
  - Erhalt der Reihenfolge von gleichwertigen Schlüsseln

Vertauschungen und Vergleiche

- Die Applikation ist wichtiger Auswahlfaktor:
  - Ein Verfahren mit vielen Vertauschungen und wenig Vergleichen ist gut geeignet für teure Vergleichsoperationen.
  - Falls die Umsortierung teuer ist, sollte man ein Verfahren mit wenig Vertauschungen wählen.

Beispiel 1: Sortiere Container nach Gewicht

- Teure Vertauschung
  (Kran, erst weitere Vertauschungen nötig)
- Günstige Vergleiche
  (Container-ID, Frachtpapiere)

Beispiel 2: Sortiere Beeren nach Vitamingehalt

- Günstige Vertauschung
  (Nehmen & Ablegen)
- Teure Vergleiche
  (Labortests, Zeit- und Kostenintensiv)

Weitere Aufgaben

Viele Aufgaben sind mit dem Sortieren verwandt und können auf das Sortierproblem zurückgeführt werden:

- Bestimmung des Median (der Median ist definiert als das Element an der mittleren Position der sortierten Folge)
- Bestimmung der k kleinsten bzw. k größten Elemente

Typen von Sortieralgorithmen

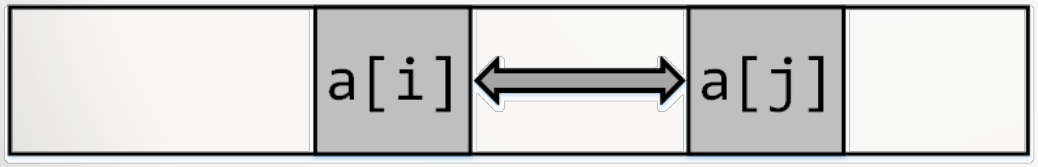
- Einfache
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - …
- Höhere
  - MergeSort
  - QuickSort
  - HeapSort
  - …
- Spezielle
  - BucketSort
  - …

Swap

Im Folgenden wird häufig der Aufruf `swap(a, i, j)` verwendet, wobei `a` ein Array ist und `i`, `j` zwei int-Werte.

Der Aufruf ersetzt folgende drei Zuweisungen:

```
Object temp = a[i];
a[i] = a[j];
a[j] = temp;
```



BubbleSort

- Idee:
  - Vergleiche Paare von benachbarten Schlüsseln
  - Tausche, falls linker Schlüssel größer ist als rechter
- Durch das repetitive Vertauschen verhalten sich Elemente wie aufsteigende Luftblasen im Wasser

```
public void bubblesort(int[] a){
  int n = a.length;
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n-i-1; j++){
      if(a[j] > a[j+1])
        swap(a,j,j+1);
    }
  }
}
```

BubbleSort: Beispiel

3	7	8	6	4	2

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	0

BubbleSort: Beispiel

3	7	8	6	4	2

$3 < 7$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	0

BubbleSort: Beispiel

3	7	8	6	4	2

$7 < 8$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	1

BubbleSort: Beispiel

3	7	8	6	4	2

$$8 \not< 6$$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	2

BubbleSort: Beispiel

3	7	6	8	4	2

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	2

BubbleSort: Beispiel

3	7	6	8	4	2

$8 \not< 4$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	3

BubbleSort: Beispiel

3	7	6	4	8	2

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	3

BubbleSort: Beispiel

3	7	6	4	8	2

$8 \not< 2$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	4

BubbleSort: Beispiel

3	7	6	4	2	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	4

BubbleSort: Beispiel

3	7	6	4	2	8

$3 < 7$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	0

BubbleSort: Beispiel

3	7	6	4	2	8

$7 \not< 6$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	1

BubbleSort: Beispiel

3	6	7	4	2	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	1

BubbleSort: Beispiel

3	6	7	4	2	8

$7 \not< 4$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	2

BubbleSort: Beispiel

3	6	4	7	2	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	2

BubbleSort: Beispiel

3	6	4	7	2	8

$7 \not< 2$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	3

BubbleSort: Beispiel

3	6	4	2	7	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	3

BubbleSort: Beispiel

3	6	4	2	7	8

$3 < 6$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	0

BubbleSort: Beispiel

3	6	4	2	7	8

$6 \not< 4$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	1

BubbleSort: Beispiel

3	4	6	2	7	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	1

BubbleSort: Beispiel

3	4	6	2	7	8

$6 \not< 2$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	2

BubbleSort: Beispiel

3	4	2	6	7	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	2

BubbleSort: Beispiel

3	4	2	6	7	8

$3 < 4$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	3
j	0

BubbleSort: Beispiel

3	4	2	6	7	8

$4 \not< 2$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	3
j	1

BubbleSort: Beispiel

3	2	4	6	7	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	3
j	1

BubbleSort: Beispiel

3	2	4	6	7	8

$3 \not< 2$

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	4
j	0

BubbleSort: Beispiel

2	3	4	6	7	8

Swap

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	4
j	0

BubbleSort: Beispiel

2	3	4	6	7	8

Sortiert!

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	4
j	0

BubbleSort: Komplexitätsanalyse Vergleiche

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

- Anzahl der Vergleiche:
  - Unabhängig von der Vorsortierung der Folge
    ⇒ worst case, average case und best case sind identisch
    (es werden stets alle Elemente der noch nicht sortierten Teilfolge miteinander verglichen)
  - Im i-ten Schleifendurchlauf werden $n - i + 1$ Elemente betrachtet und $n - i$ Paare verglichen, insgesamt:

$$\sum_{i=1}^{n}(n-i) = \sum_{i=0}^{n-1} i = O(n^2)$$

BubbleSort: Komplexitätsanalyse Vertauschungen

```
int n = a.length;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

- Anzahl der Vertauschungen:
  - Best Case: 0 Swaps (Vorsortierung)
  - Worst Case: $\frac{1}{2}(n^2 - n)$ Swaps (Invertierte Reihenfolge)
  - Average Case: $\frac{1}{4}(n^2 - n)$ Swaps
    - » Donald E. Knuth: *The Art of Computer Programming (1997)*

SelectionSort

- Idee: Suche jeweils nächstes kleinstes Element
  - Durchlaufe die Folge von links nach rechts mit einem Zeiger i.
  - Links von i sind die i-kleinsten Elemente sortiert.
  - Finde in der verbleibenden Menge das kleinste Element und tausche es mit i. Dann erhöhe i um eins.
  - Tausche, falls linker Schlüssel größer ist als rechter

```
public void selectionsort(int[] a){
  for(int i = 0; i < a.length-1; i++) {
    int min = i;
    for(int j = i+1; j < n; j++){
      if(a[j] < a[min])
        min = j;
    }
    swap(a,i,min);
  }
}
```

SelectionSort: Beispiel

3	7	8	6	4	2

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	1
min	0

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	1
min	0

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	2
min	0

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

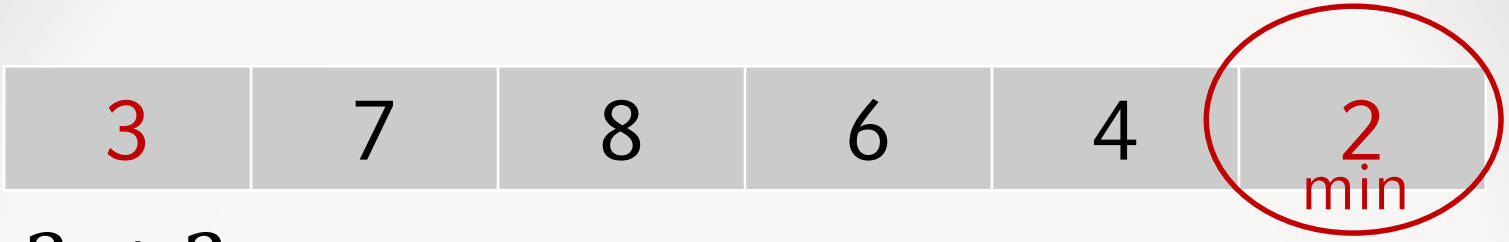
i	0
j	3
min	0

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	4
min	0

SelectionSort: Beispiel



$3 \not< 2$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	5
min	5

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

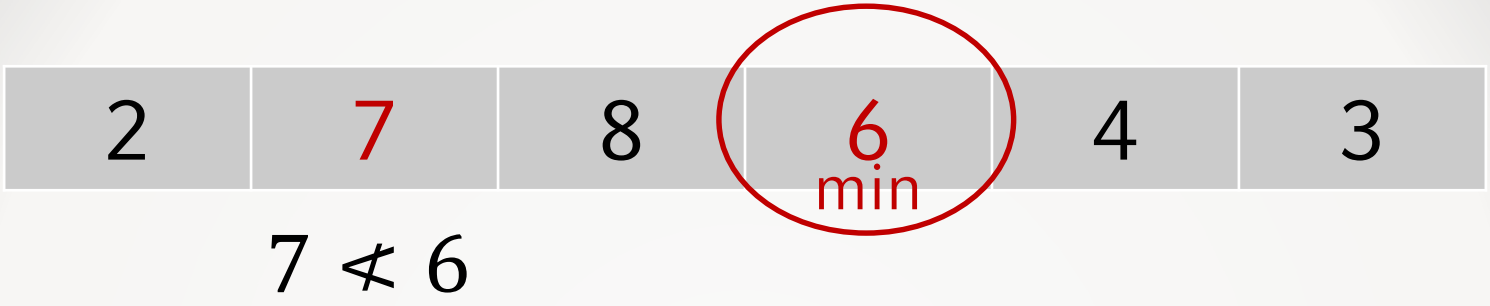
i	0
j	5
min	5

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	2
min	1

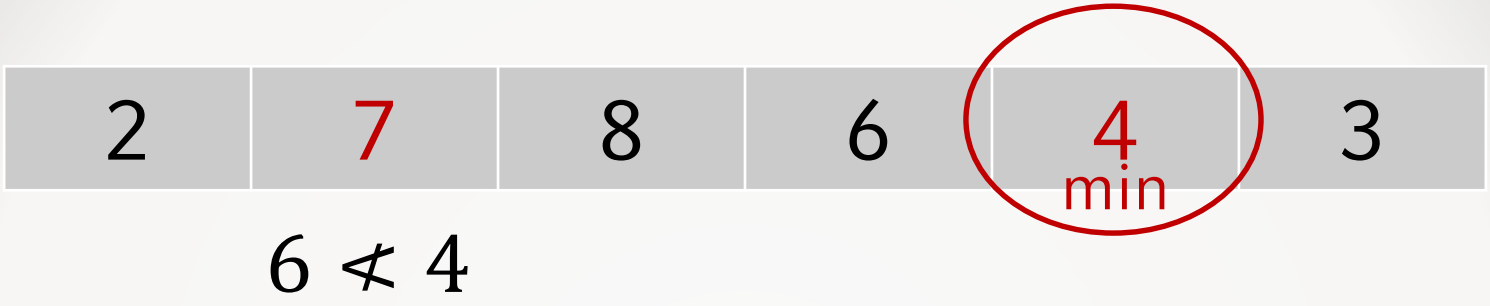
SelectionSort: Beispiel




```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	3
min	3

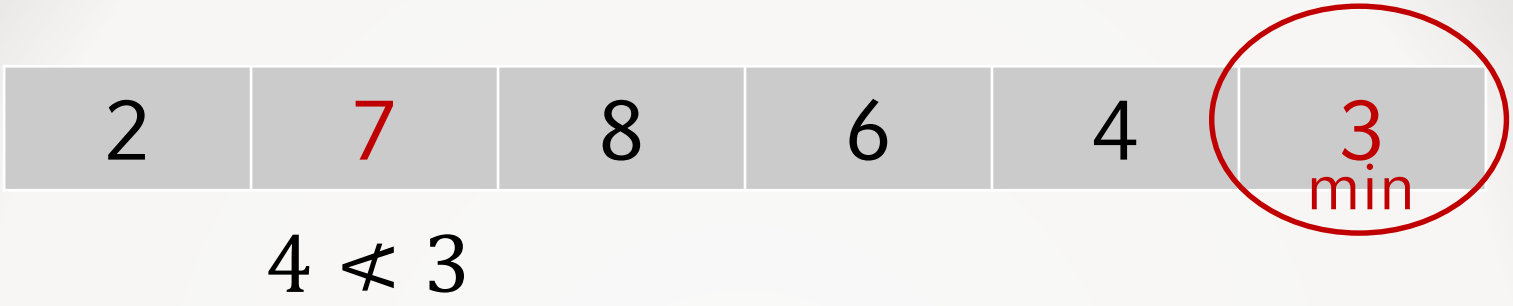
SelectionSort: Beispiel




```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	4
min	4

SelectionSort: Beispiel




```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

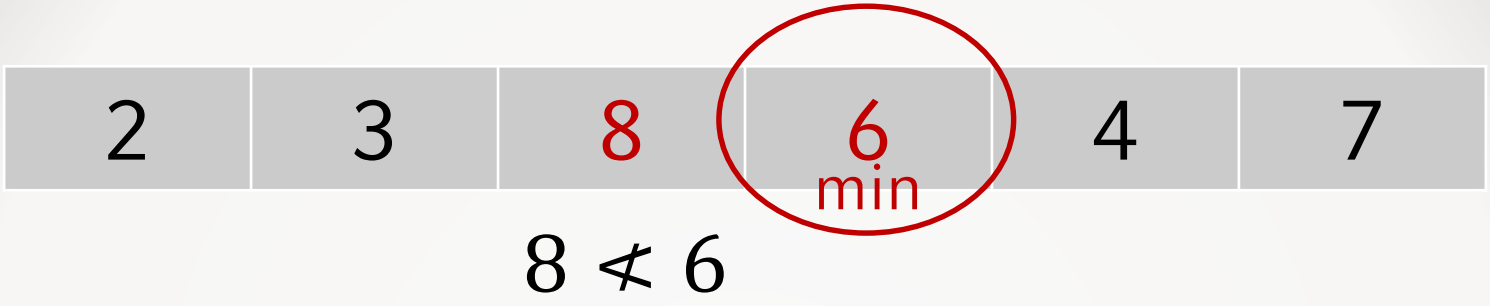
i	1
j	5
min	5

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	5
min	5

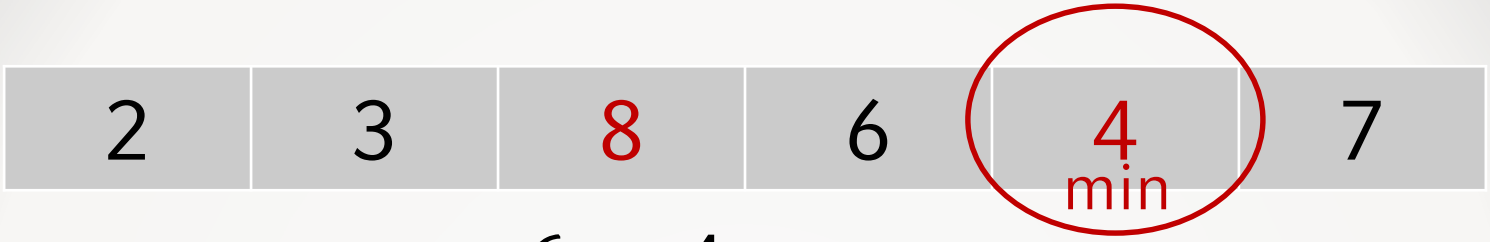
SelectionSort: Beispiel




```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	3
min	3

SelectionSort: Beispiel

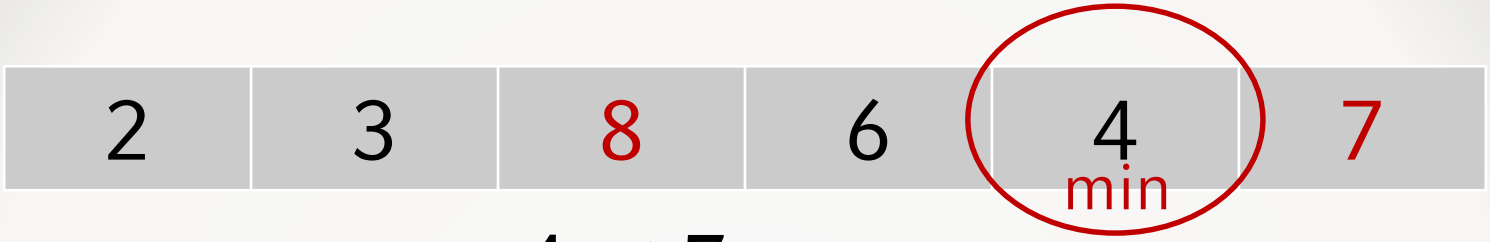


$6 \not< 4$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	4
min	4

SelectionSort: Beispiel



$4 < 7$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	5
min	4

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	5
min	4

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

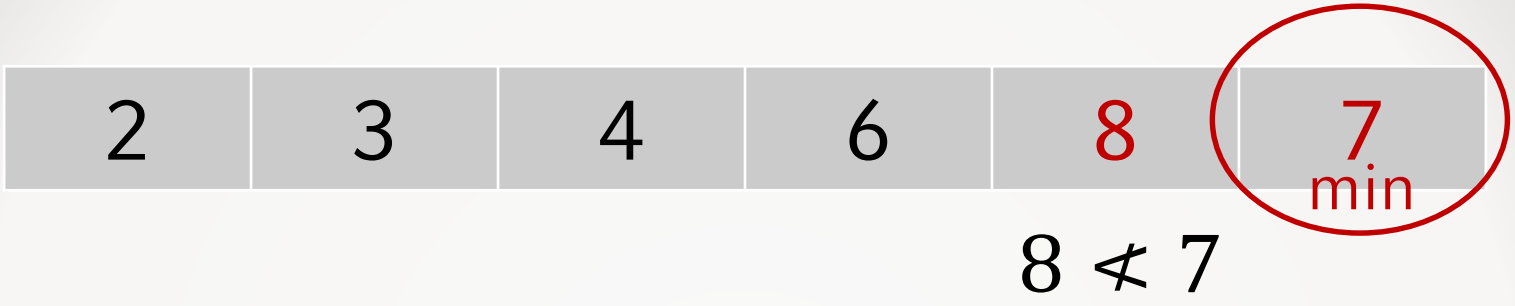
i	3
j	4
min	3

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	3
j	5
min	3

SelectionSort: Beispiel




```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	4
j	5
min	5

SelectionSort: Beispiel


```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	4
j	5
min	5

SelectionSort: Beispiel

2	3	4	6	7	8

Sortiert!

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	4
j	5
min	5

SelectionSort: Komplexitätsanalyse Vergleiche

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

- Anzahl der Vergleiche:
  - In jedem der n Durchläufe $n - i$ Vergleiche

$$\Rightarrow \frac{n(n-1)}{2} \in O(n^2)$$

  - Durch weitere if-Abfrage (min == i) lässt sich Anzahl der swaps im Austausch gegen n zusätzliche Vergleiche verringern

SelectionSort: Komplexitätsanalyse Vertauschungen

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

- Anzahl der Vertauschungen:
  - Best Case: $n - 1$ Swaps (0 mit zusätzlicher Abfrage)
  - Worst Case: $n - 1$ Swaps
  - Average Case: $n - 1$ Swaps
- Anzahl Swaps wachsen linear.
  Daher ist SelectionSort besonders für Folgen großer Objekte geeignet, deren Vertauschungen teurer sind.

InsertionSort

- Idee:
  - Halte die linke Teilfolge sortiert.
  - Füge nächstes Objekt hinzu, indem es an die korrekte Position eingefügt wird.
  - Wiederhole dies, bis Teilfolge aus der gesamten Liste besteht.

```
public void insertionsort(int[] a){
  for(int i = 1; i < a.length; i++) {
    int key = a[i];
    int j = i;
    while(j > 0 && a[j-1] > key){
      a[j] = a[j-1];
      j--;
    }
    a[j] = key;
  }
}
```

InsertionSort: Beispiel

3	7	8	6	4	2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	1
j	1

InsertionSort: Beispiel

3	7	8	6	4	2

$3 < 7$

7

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	1
j	1

InsertionSort: Beispiel

3	7	8	6	4	2

$7 < 8$

8

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	2
j	2

InsertionSort: Beispiel

3	7	8	6	4	2

$8 \not< 6$

6

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	3

InsertionSort: Beispiel

3	7	8	8	4	2

$7 \not< 6$

6

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	2

InsertionSort: Beispiel

3	7	7	8	4	2

$3 < 6$

6

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	1

InsertionSort: Beispiel

3	6	7	8	4	2

insert

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	1

InsertionSort: Beispiel

3	6	7	8	4	2

$4 < 8$

4

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	4

InsertionSort: Beispiel

3	6	7	8	8	2

$4 < 7$

4

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	3

InsertionSort: Beispiel

3	6	7	7	8	2

$4 < 6$

4

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	2

InsertionSort: Beispiel

3	6	6	7	8	2

$4 \not< 3$

4

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	1

InsertionSort: Beispiel

3	4	6	7	8	2

insert

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	1

InsertionSort: Beispiel

3	4	6	7	8	2

$2 < 8$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	5

InsertionSort: Beispiel

3	4	6	7	8	8
			$2 < 7$	2	

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	4

InsertionSort: Beispiel

3	4	6	7	7	8

$2 < 6$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	3

InsertionSort: Beispiel

3	4	6	6	7	8

$2 < 4$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	2

InsertionSort: Beispiel

3	4	4	6	7	8

$2 < 3$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	1

InsertionSort: Beispiel

3	3	4	6	7	8

$j = 0$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	0

InsertionSort: Beispiel

2	3	4	6	7	8

insert

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	0

InsertionSort: Beispiel

2	3	4	6	7	8

Sortiert!

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	0

InsertionSort: Variante mit Sentinel (= Wächterelement)

- Erweitere das Eingabearray vorne um eine Position

$-\infty$	2	3	4	6	7	8

Wächter

- Der Code kann nun auf eine Abfrage verzichten:

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

InsertionSort: Komplexitätsanalyse Vergleiche

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

- Anzahl der Vergleiche:
  - Abhängig von Vorsortierung
  - Im Best Case $O(n)$ Vergleiche bei sortierter Folge
  - Average Case und Worst Case: $O(n^2)$

InsertionSort: Komplexitätsanalyse Verschiebeoperationen

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

- Im besten Fall ist die Folge sortiert und man benötigt keine Verschiebung
- Im schlimmsten Fall ist die Folge absteigend sortiert und das j-te Element wird mit $j-1$ Operationen an den jeweiligen Anfang geschoben: $\frac{n(n-1)}{2} \in O(n^2)$
- Man kann zeigen, dass im Average Case $\frac{n(n-1)}{4} \in O(n^2)$ Operationen nötig sind.

Fazit: Einfache Sortieralgorithmen

- BubbleSort:
  - Einfache Implementierung.
  - Führt auch im Best Case / Average Case viele Vergleiche aus.
- SelectionSort:
  - Benötigt durchschnittlich weniger Vertauschungen
  - Besser geeignet für teure Vertauschungen und günstige Vergleiche
- InsertionSort:
  - Mehr Vertauschungen, dafür weniger Vergleiche
  - Eignet sich gut, falls die Folge während der Ausführung erweitert wird (Datenstrom)
- Bislang benötigen alle Algorithmen $O(n^2)$ Vergleiche oder Bewegungen.

Geht es auch besser?

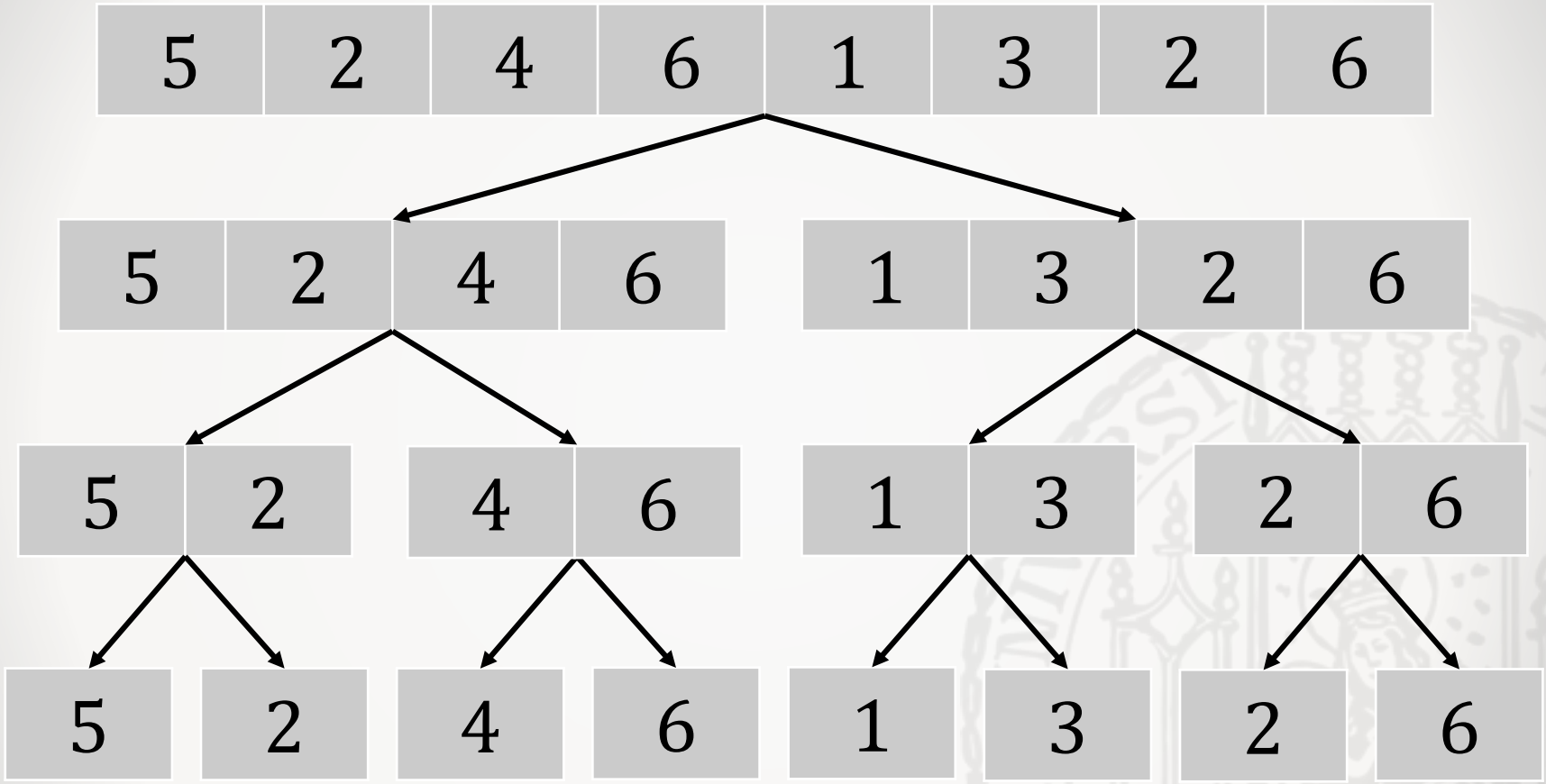
MergeSort

- Idee:
  - Zerlege die Eingabefolge in zwei gleichlange Folgen. Diese lassen sich dann unabhängig vorsortieren.
  - Anschließend werden beide Teilfolgen zusammengefügt.
  - Rekursiv können diese Schritte auch auf die Teillisten angewendet werden.
- Diese Strategie ist bekannt als Divide-and-Conquer
  - **Divide**: Zerlegen der Folge $F_{1..n} = F_{1..m} \circ F_{m+1..n}$ mit $m = \left\lfloor \frac{n}{2} \right\rfloor$:

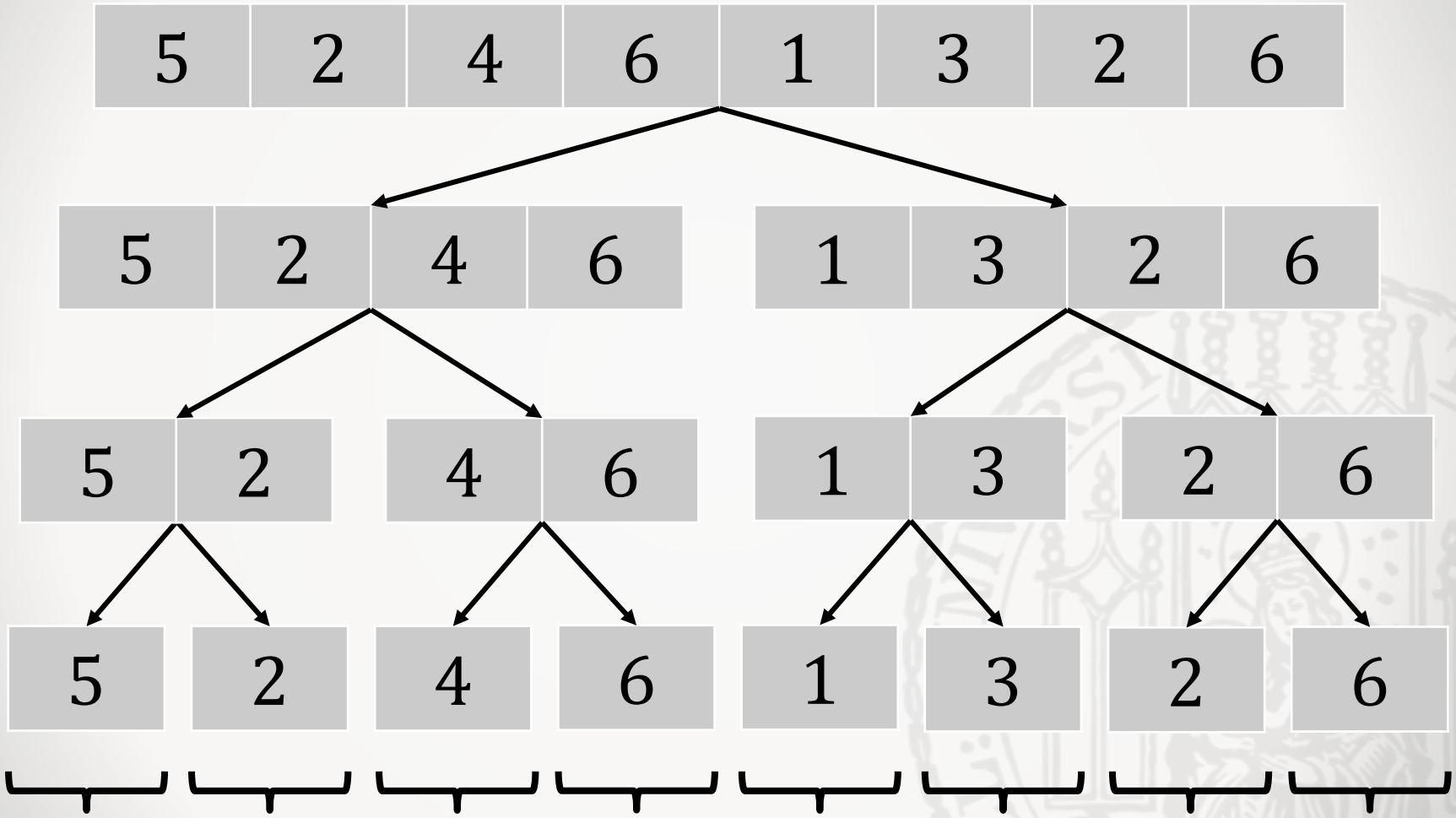
    $$F = \left(a[1], \ldots, a\left[\left\lfloor \frac{n}{2} \right\rfloor\right]\right) \circ \left(a\left[\left\lfloor \frac{n}{2} \right\rfloor + 1\right], \ldots, a[n]\right) = F_L \circ F_R$$

  - **Conquer**: Sortiere F_L und F_R mittels MergeSort, falls $|F_L| > 1$ bzw. falls $|F_R| > 1$ (sonst: einelementig, ist schon sortiert).
  - **Merge**: Verschmelze sortierte Teilfolgen F_L und F_R zu sortierter Gesamtfolge

MergeSort: Divide-Schritt

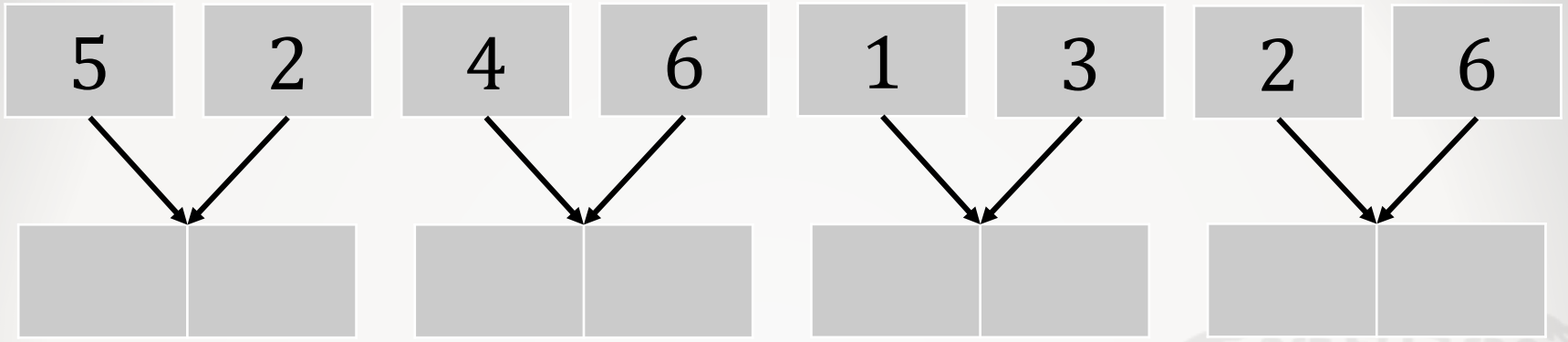


MergeSort: Conquer-Schritt

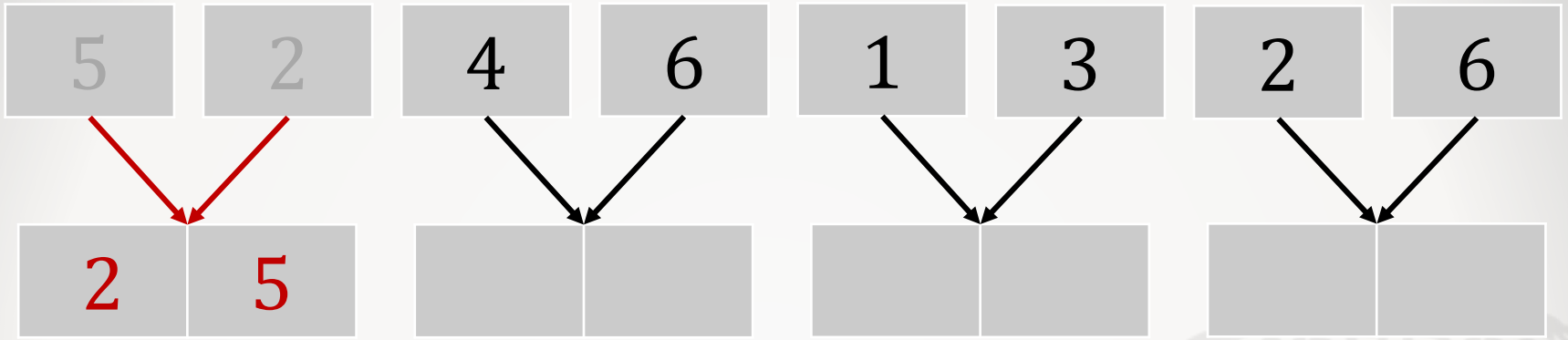


Atomare (= einelementige) Folgen sind immer sortiert

MergeSort: Merge

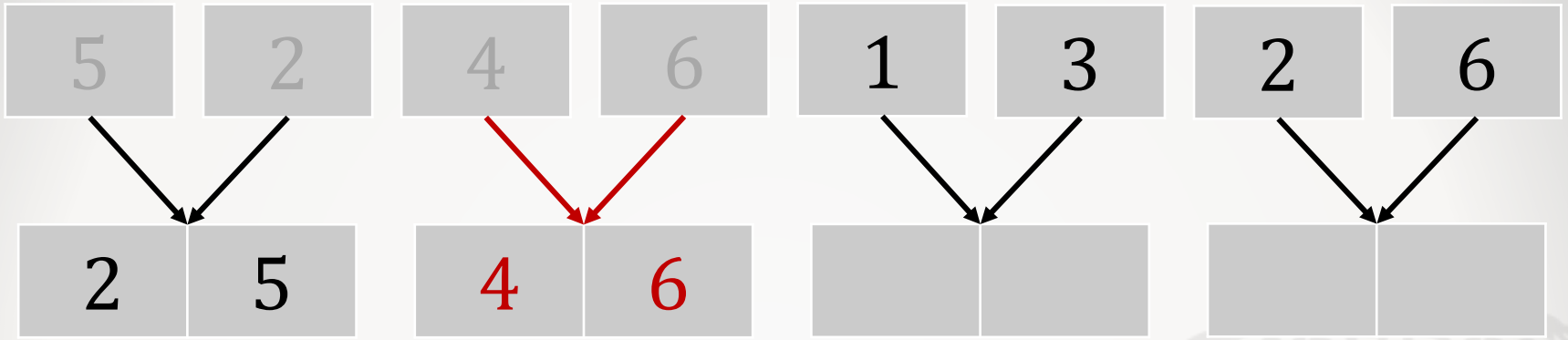


MergeSort: Merge



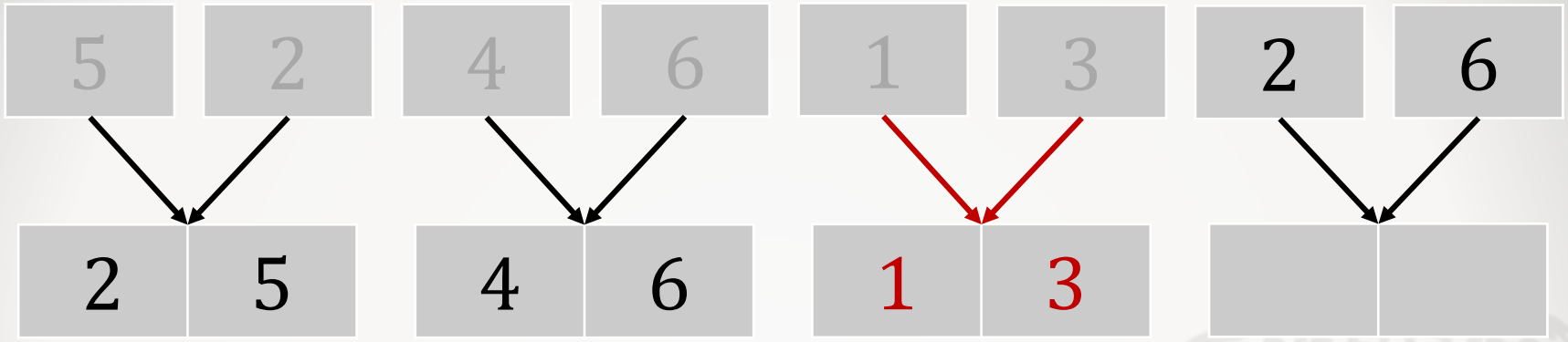
Vergleiche:
$5 < 2$?

MergeSort: Merge



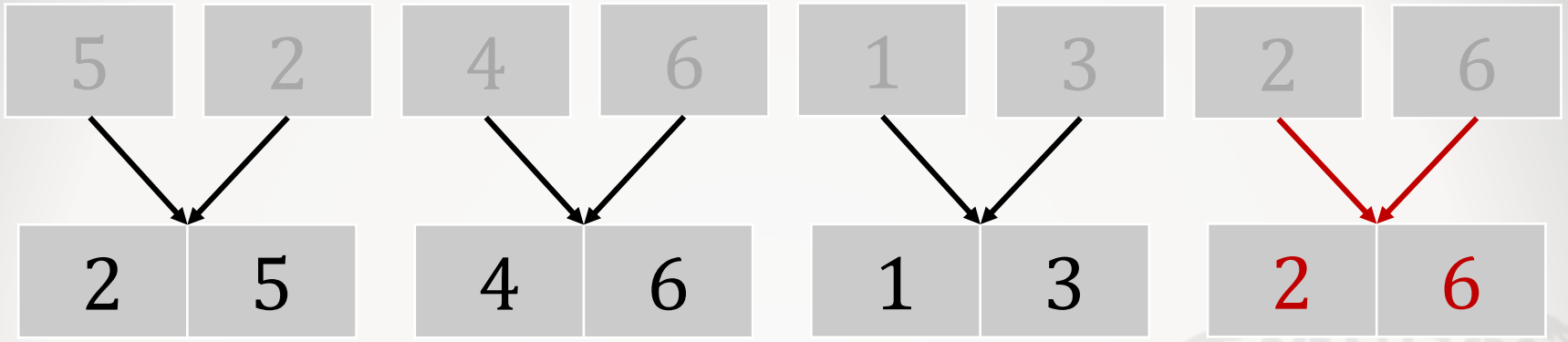
Vergleiche:
$5 < 2?$ $4 < 6?$

MergeSort: Merge



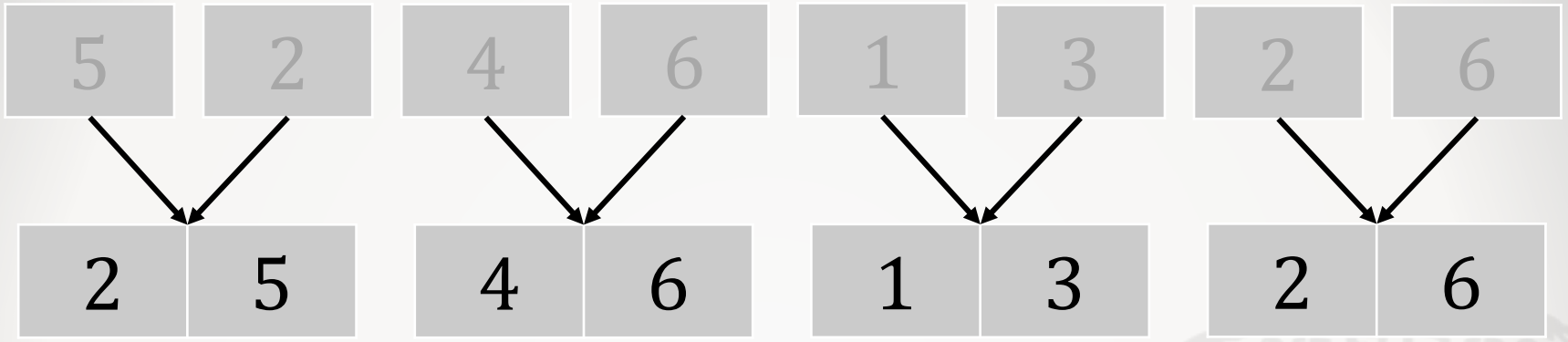
Vergleiche:
$5 < 2?$ $4 < 6?$ $1 < 3?$

MergeSort: Merge



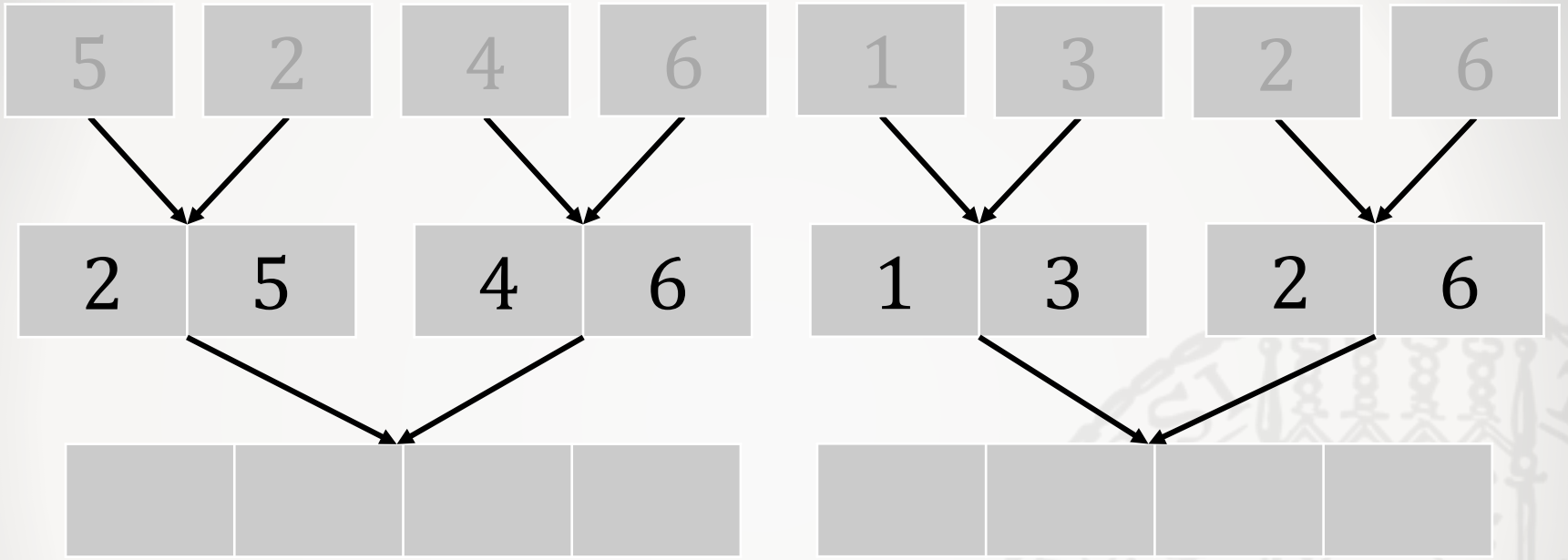
Vergleiche:
$5 < 2?\ 4 < 6?\ 1 < 3?\ 2 < 6?$

MergeSort: Merge



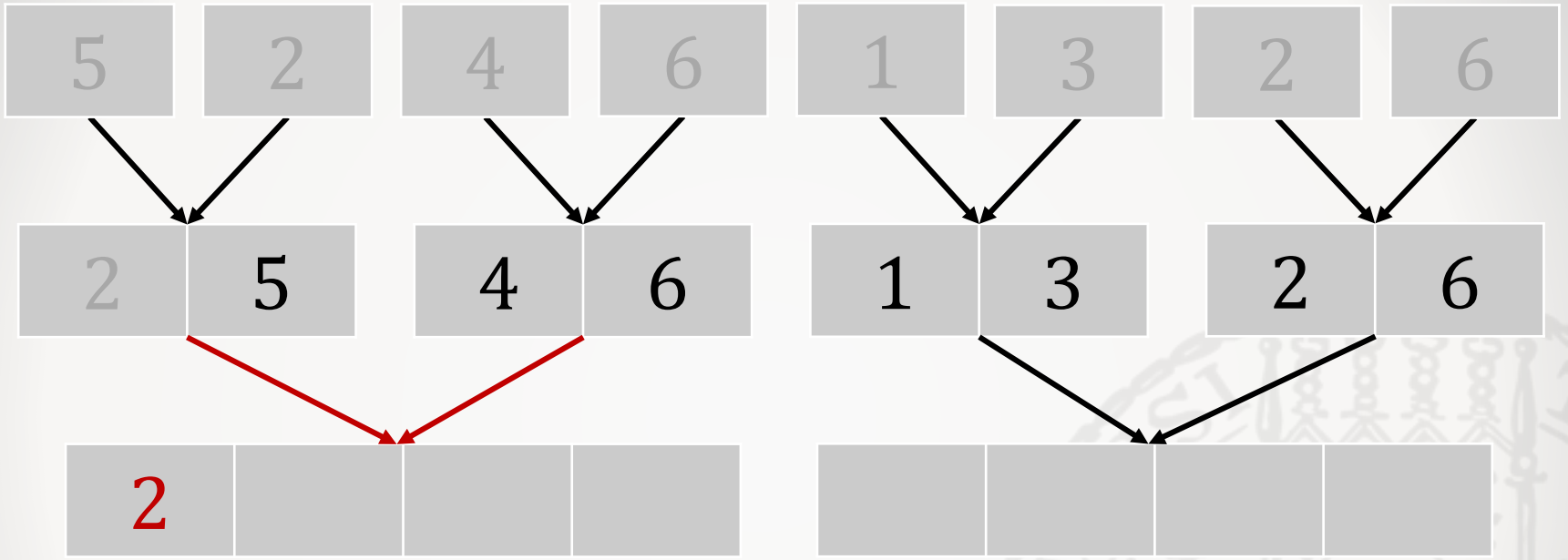
Vergleiche:
$5 < 2?\ 4 < 6?\ 1 < 3?\ 2 < 6? \Rightarrow 4$ Vergleiche

MergeSort: Merge



Vergleiche: 4

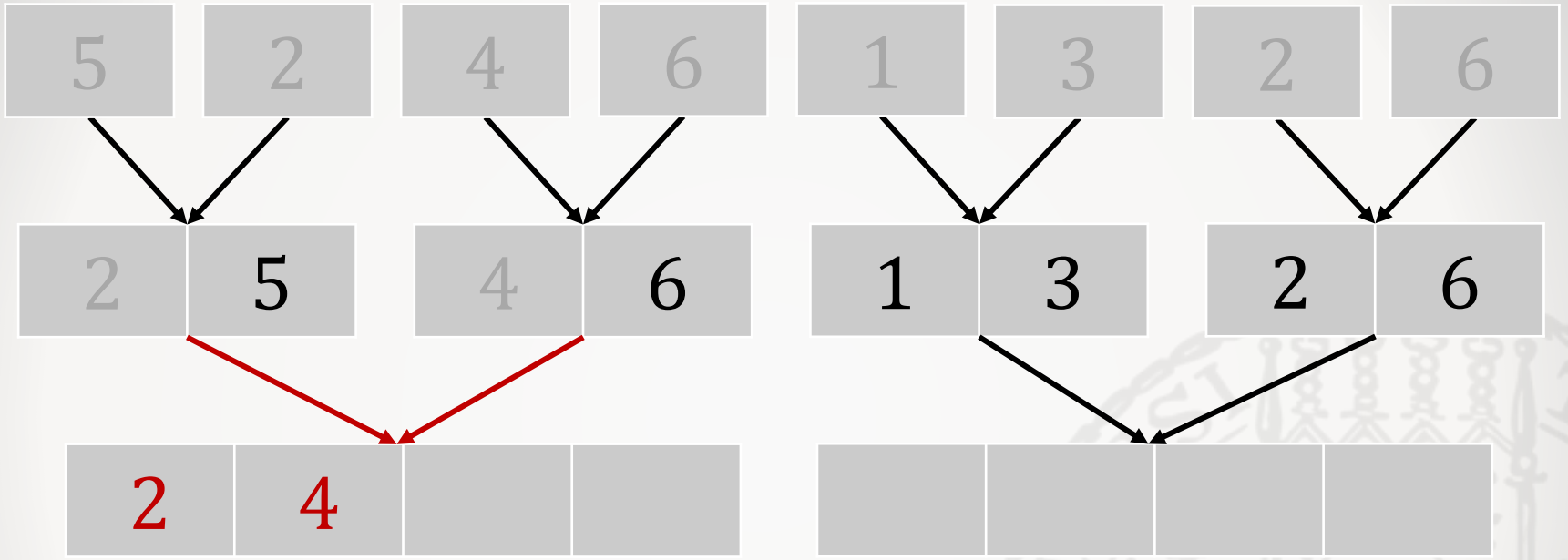
MergeSort: Merge



Vergleiche: 4

$2 < 4$?

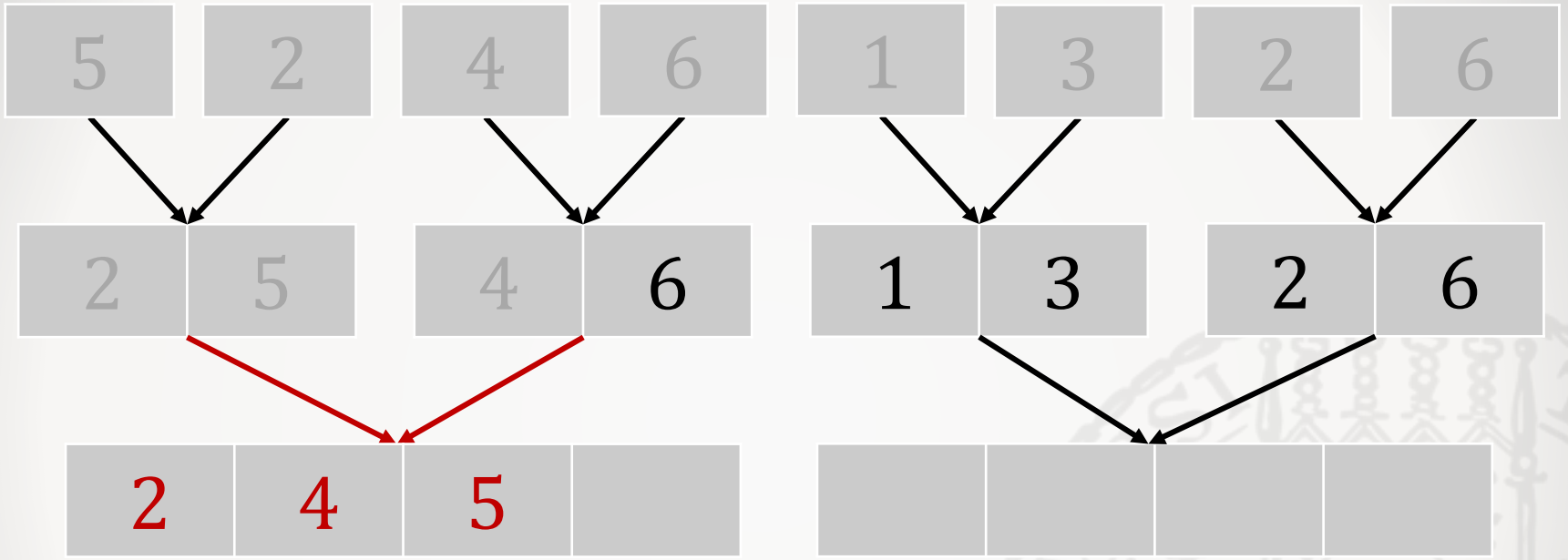
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$

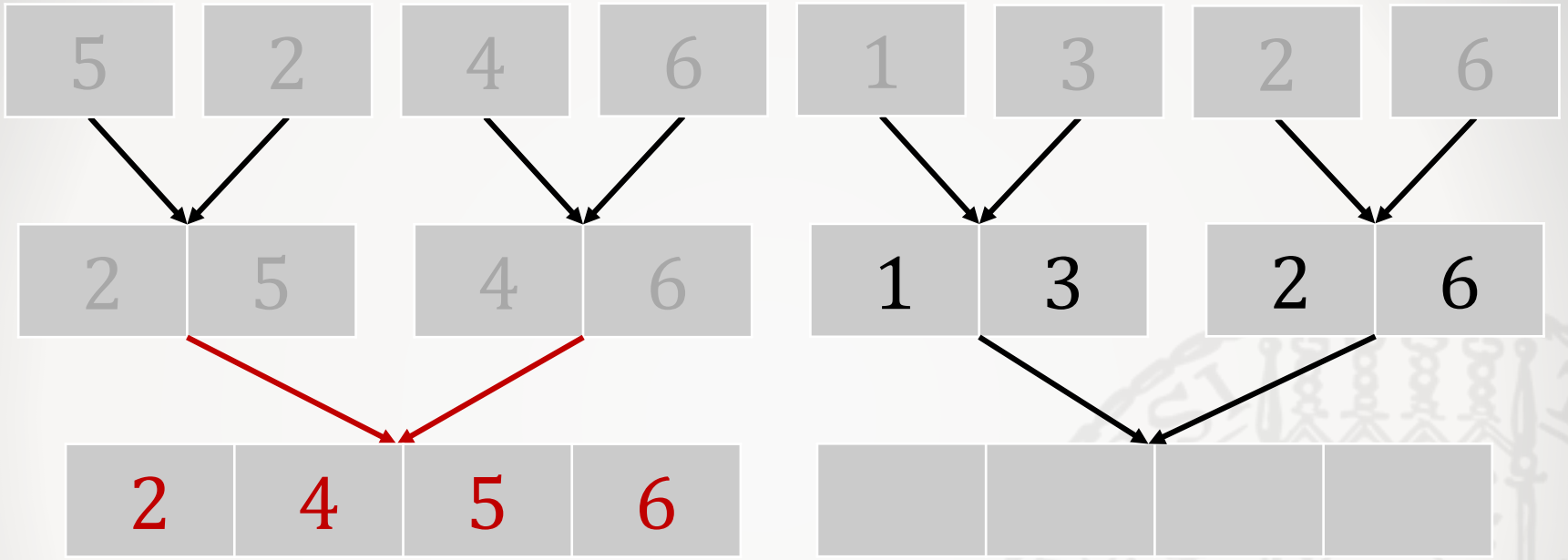
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$ $5 < 6?$

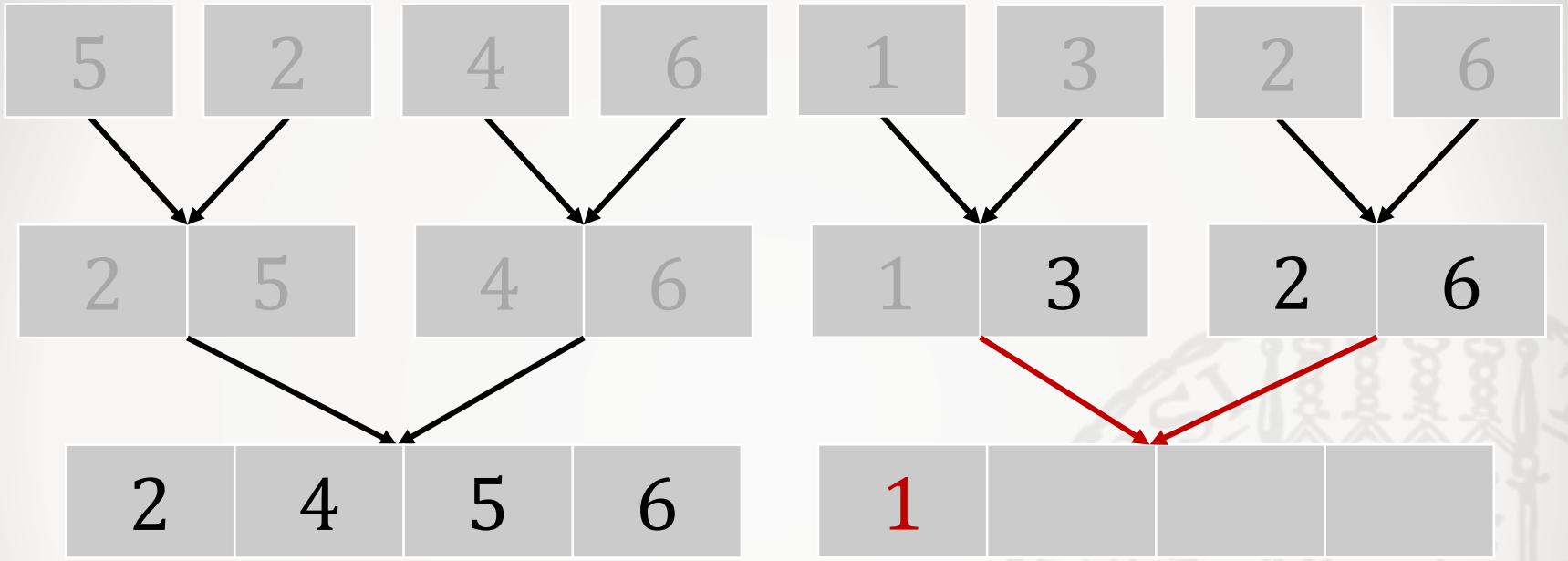
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$ $5 < 6?$

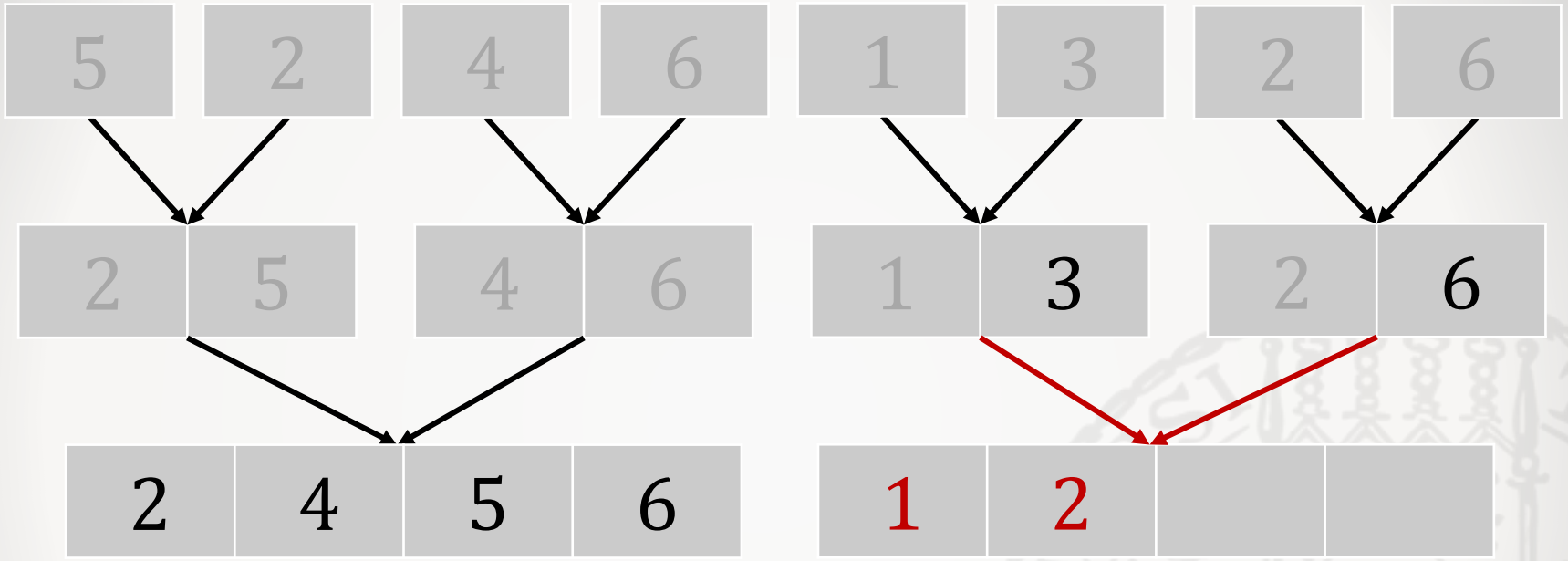
MergeSort: Merge



Vergleiche: 4

$2 < 4?\ 5 < 4?\ 5 < 6?\ 1 < 2?$

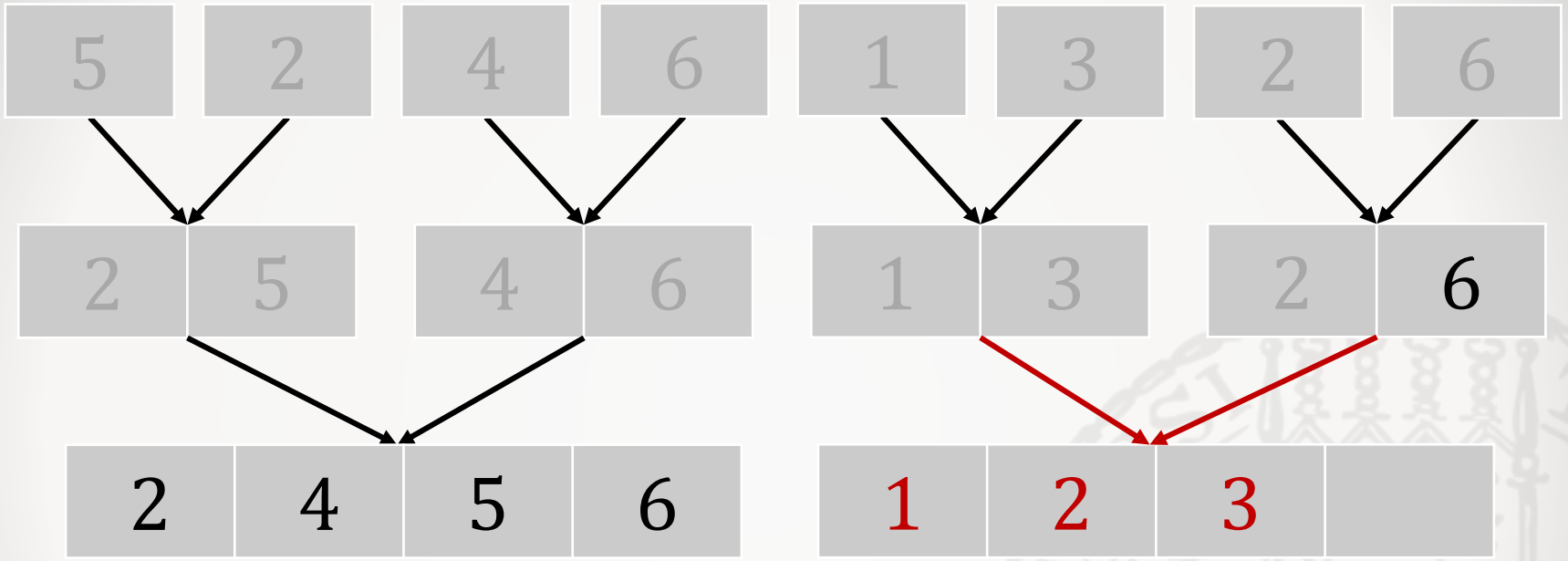
MergeSort: Merge



Vergleiche: 4

$2 < 4?\ 5 < 4?\ 5 < 6?\ 1 < 2?\ 3 < 2?$

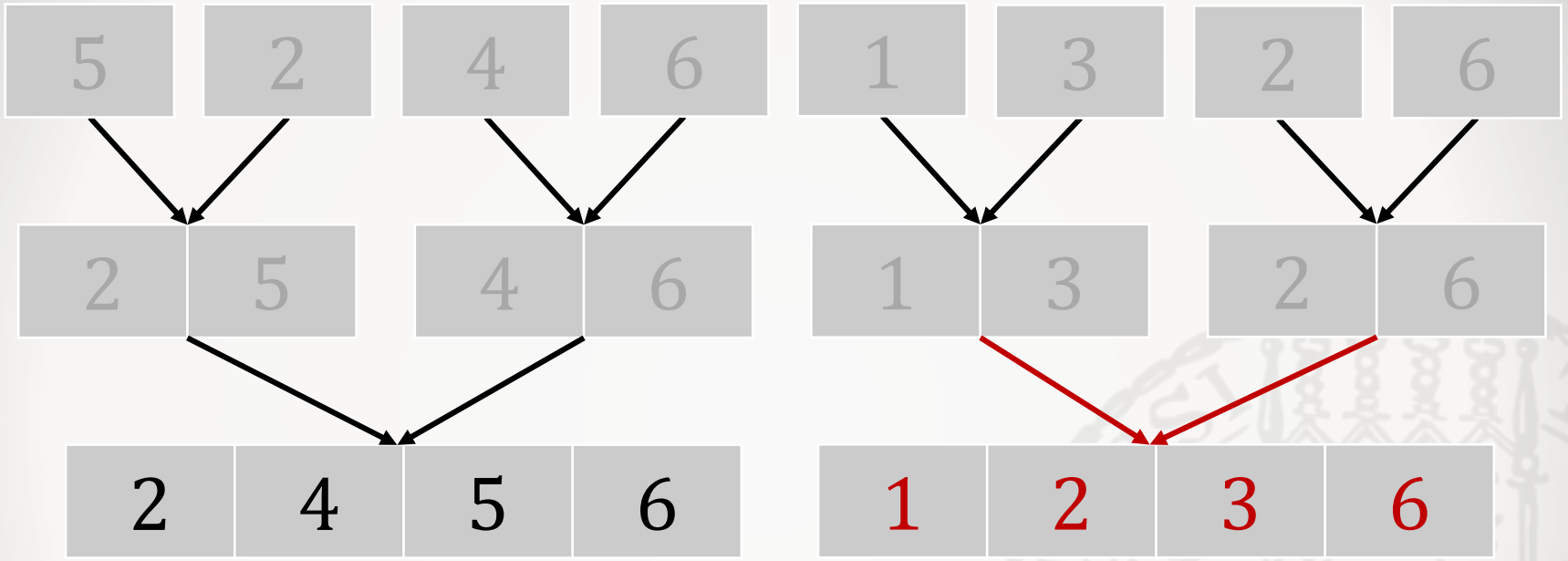
MergeSort: Merge



Vergleiche: 4

$2 < 4?\ 5 < 4?\ 5 < 6?\ 1 < 2?\ 3 < 2?\ 3 < 6?$

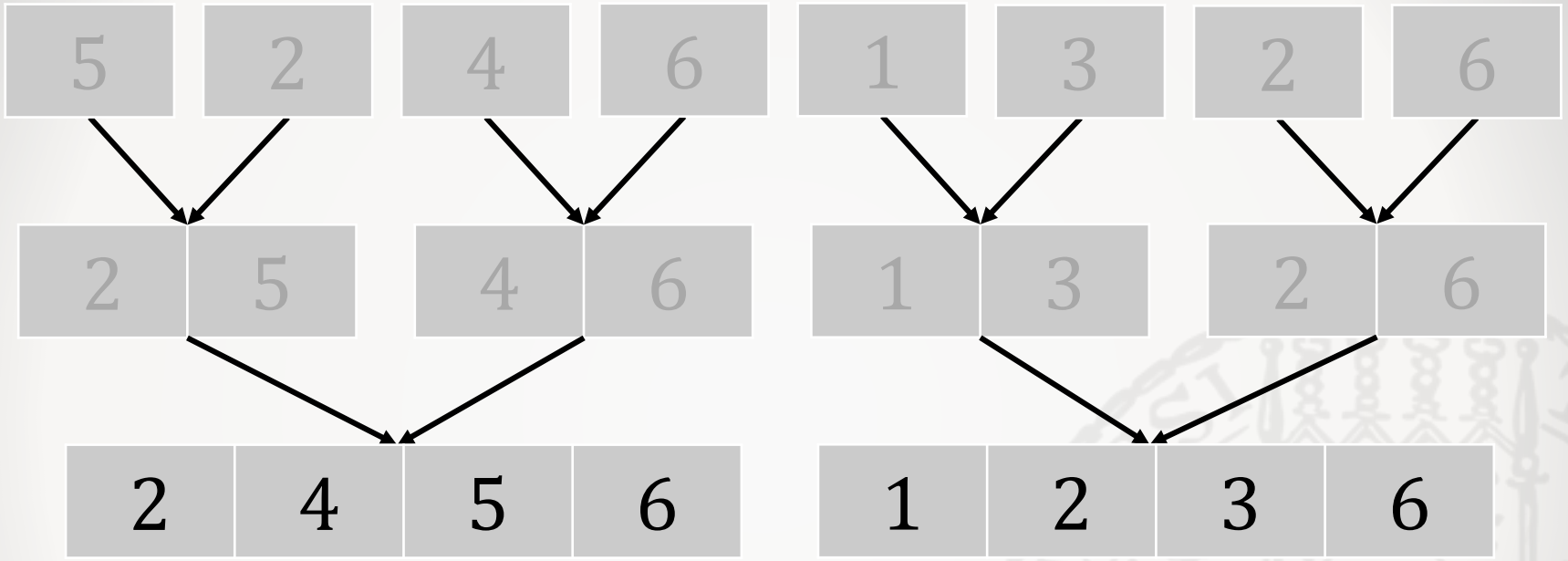
MergeSort: Merge



Vergleiche: 4

$2 < 4?\ 5 < 4?\ 5 < 6?\ 1 < 2?\ 3 < 2?\ 3 < 6?$

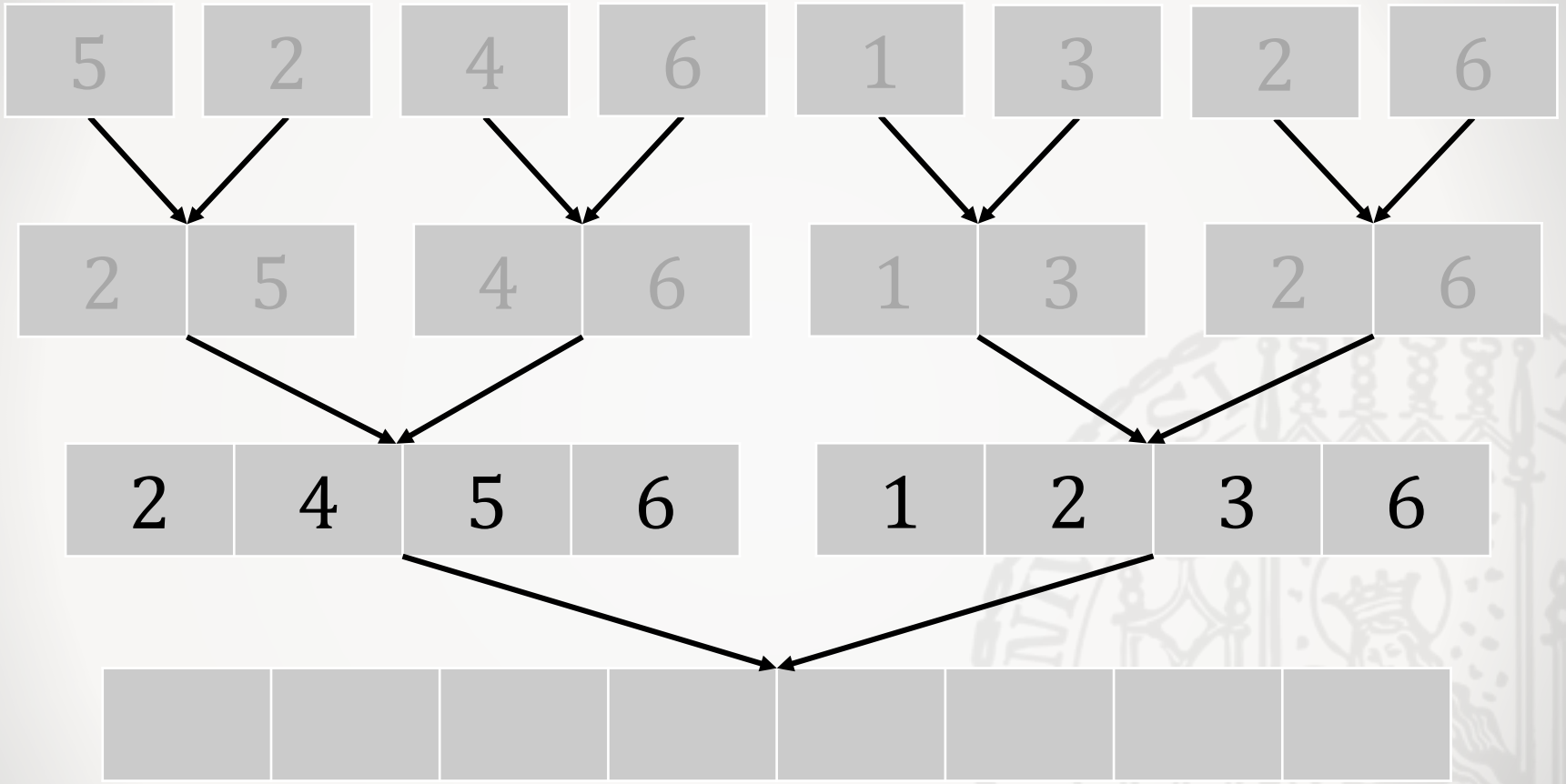
MergeSort: Merge



Vergleiche: $4 + 6$

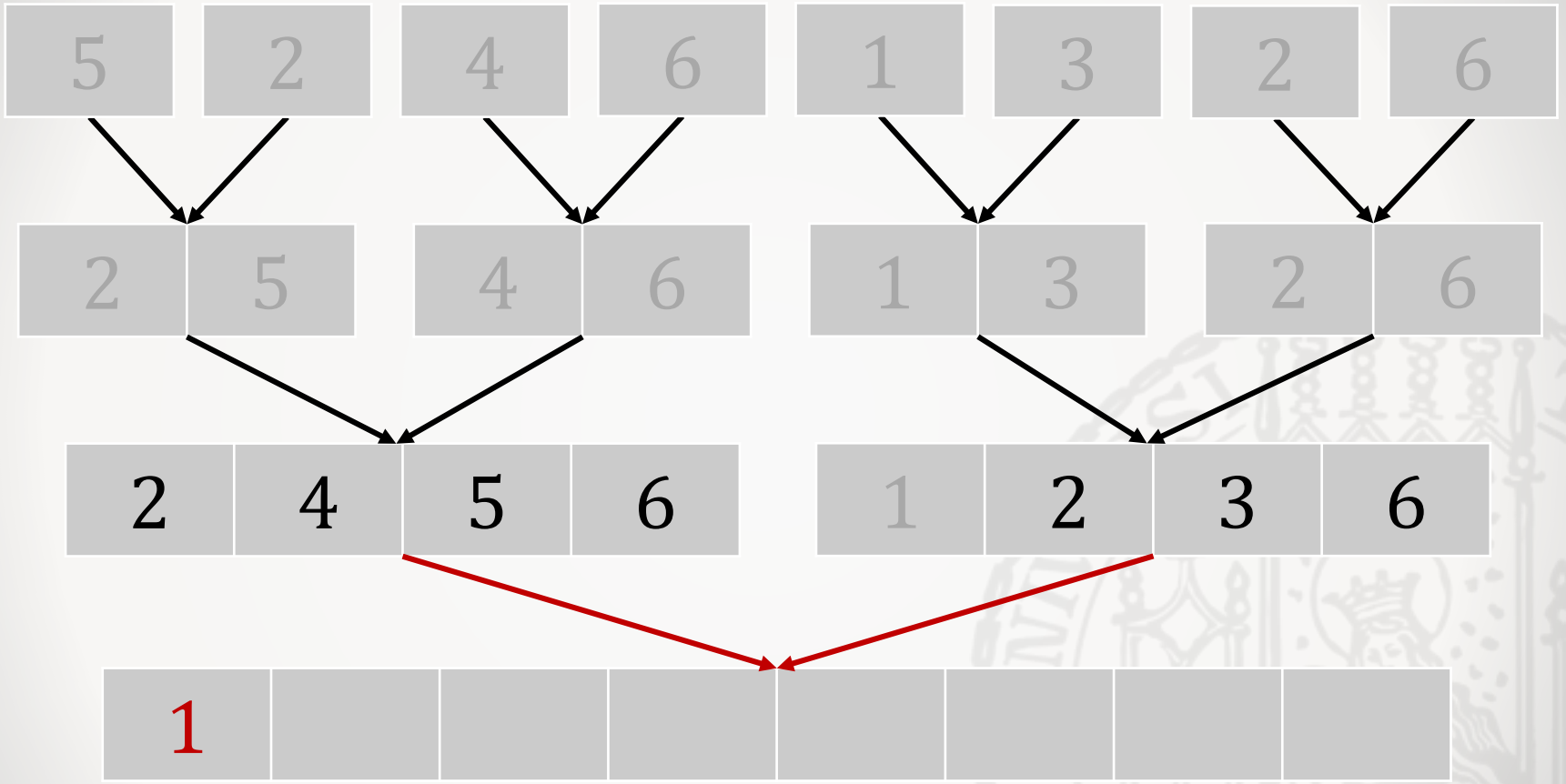
$2 < 4?\ 5 < 4?\ 5 < 6?\ 1 < 2?\ 3 < 2?\ 3 < 6? \Rightarrow 6$ Vergleiche

MergeSort: Merge



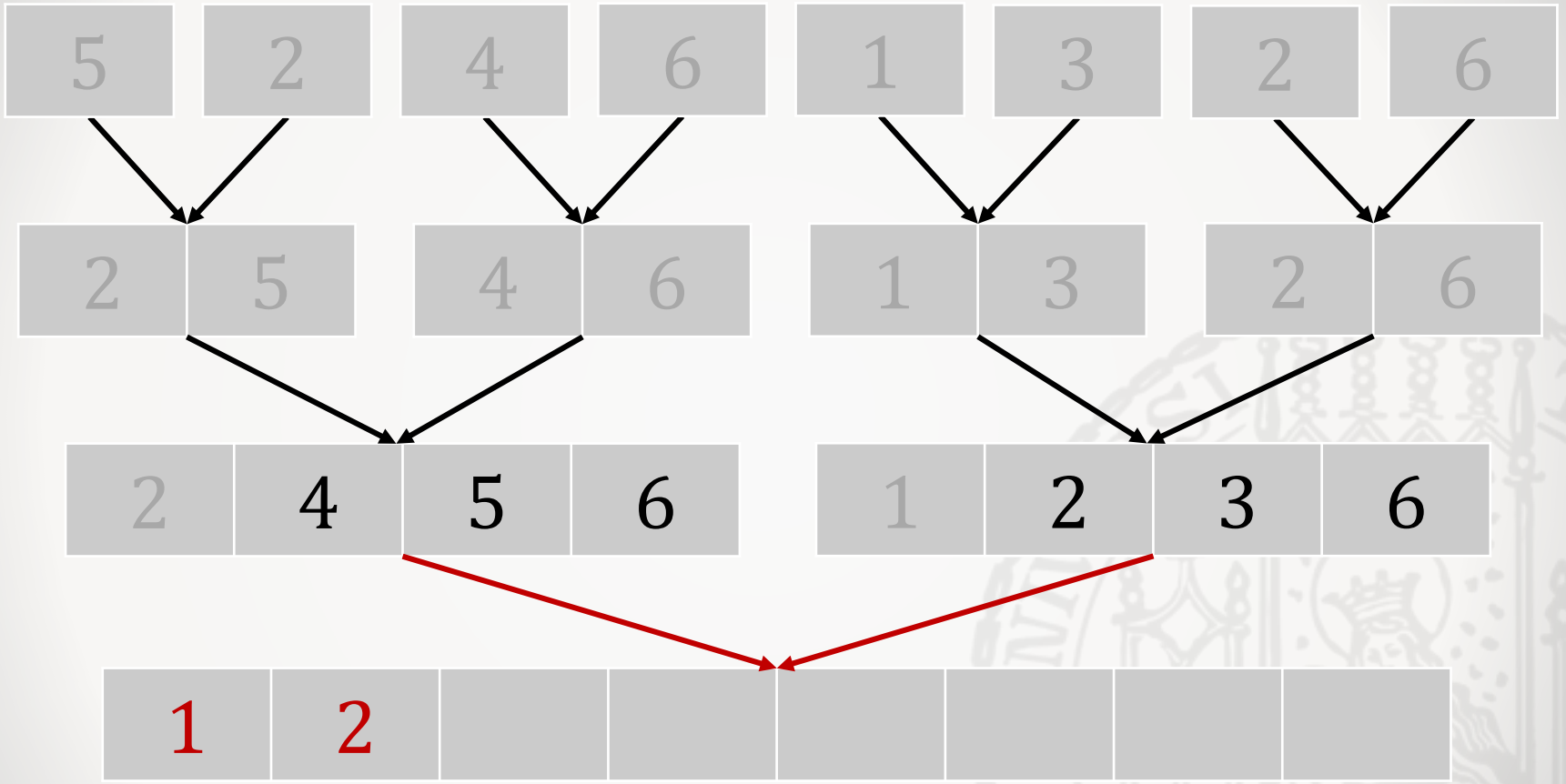
Vergleiche: 4 + 6

MergeSort: Merge



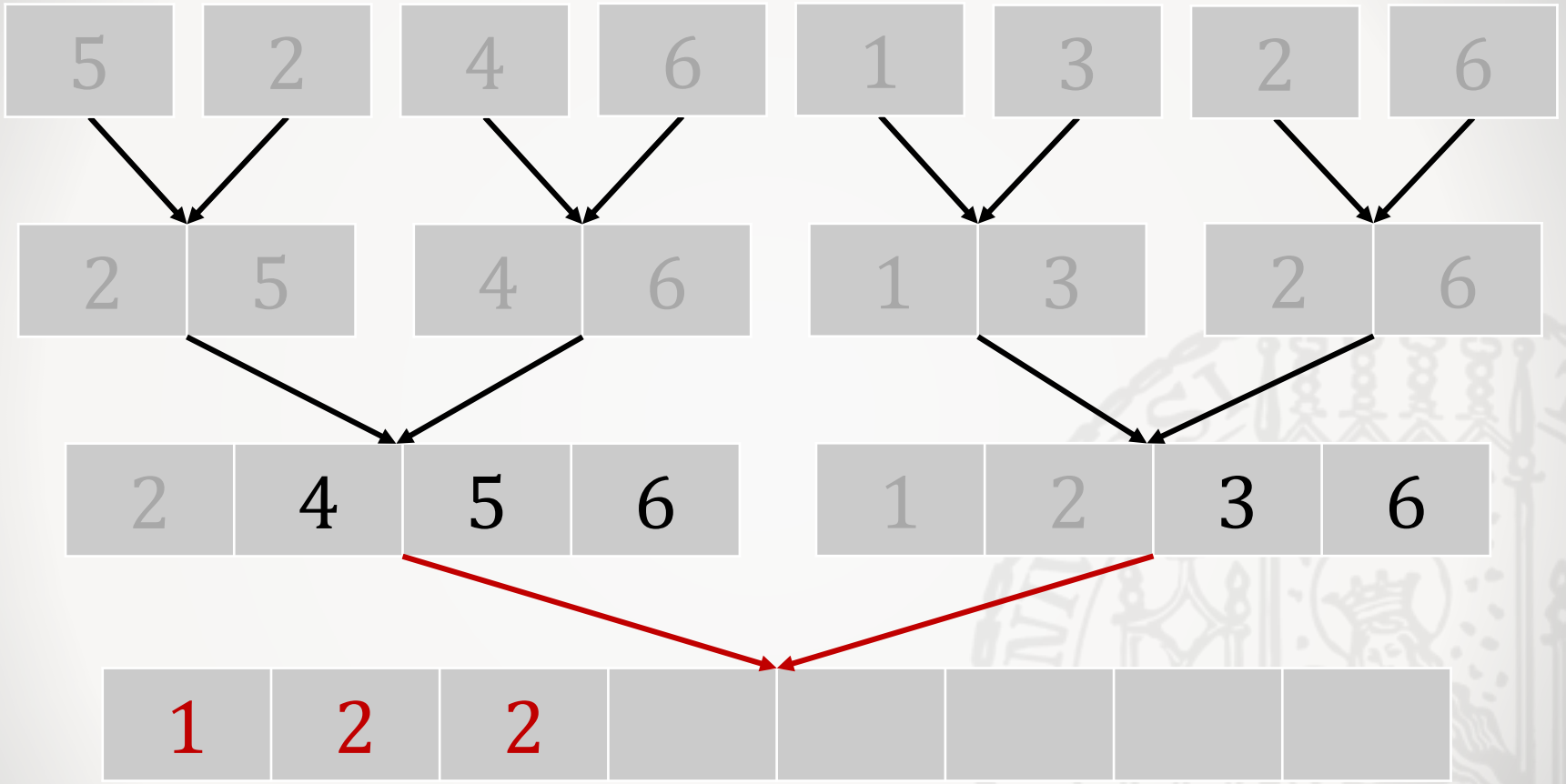
Vergleiche: $4 + 6$
$2 < 1$?

MergeSort: Merge



Vergleiche: $4 + 6$
$2 < 1?\ 2 < 2?$

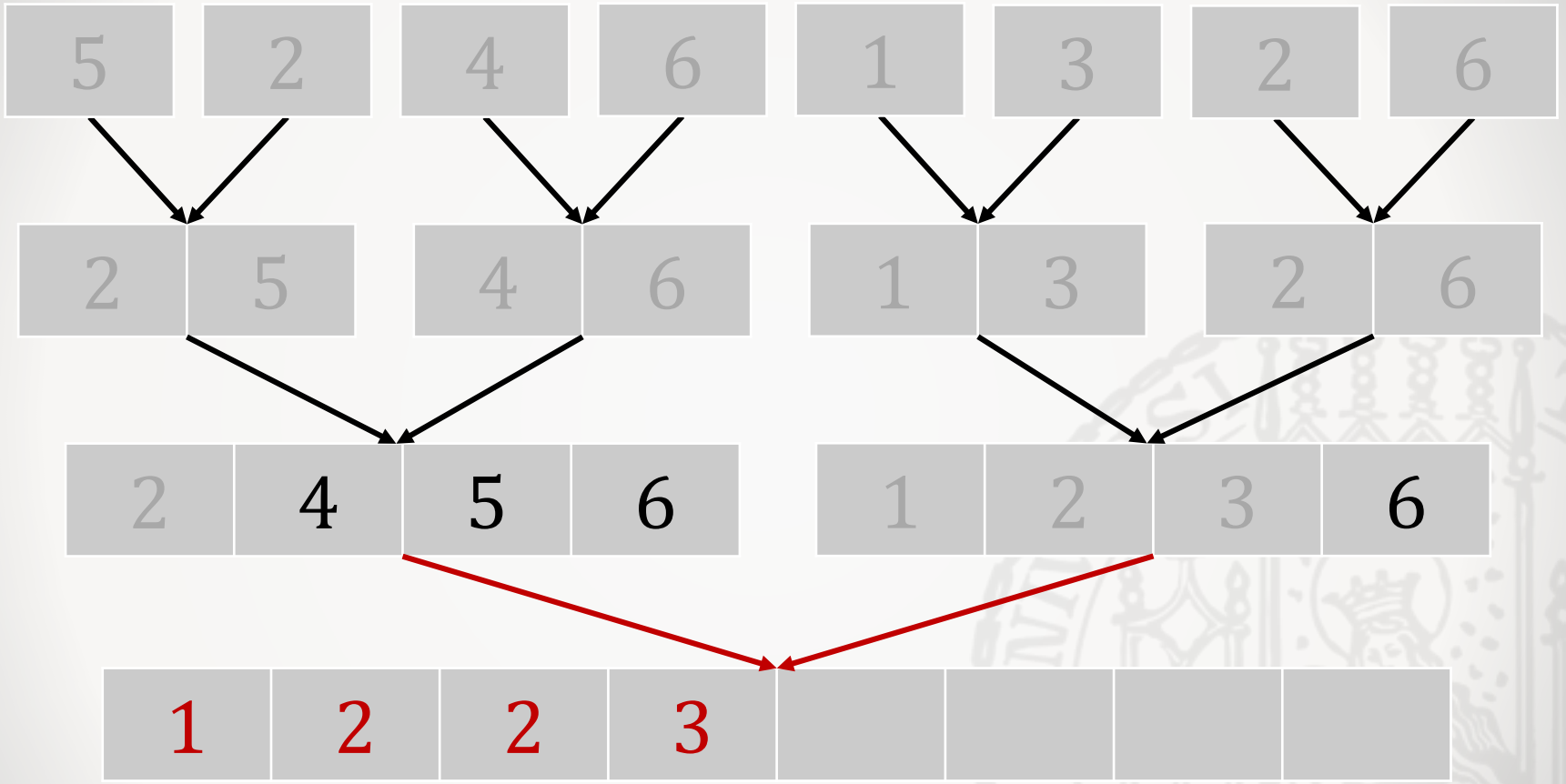
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?\ \ 2 < 2?\ \ 4 < 2?$

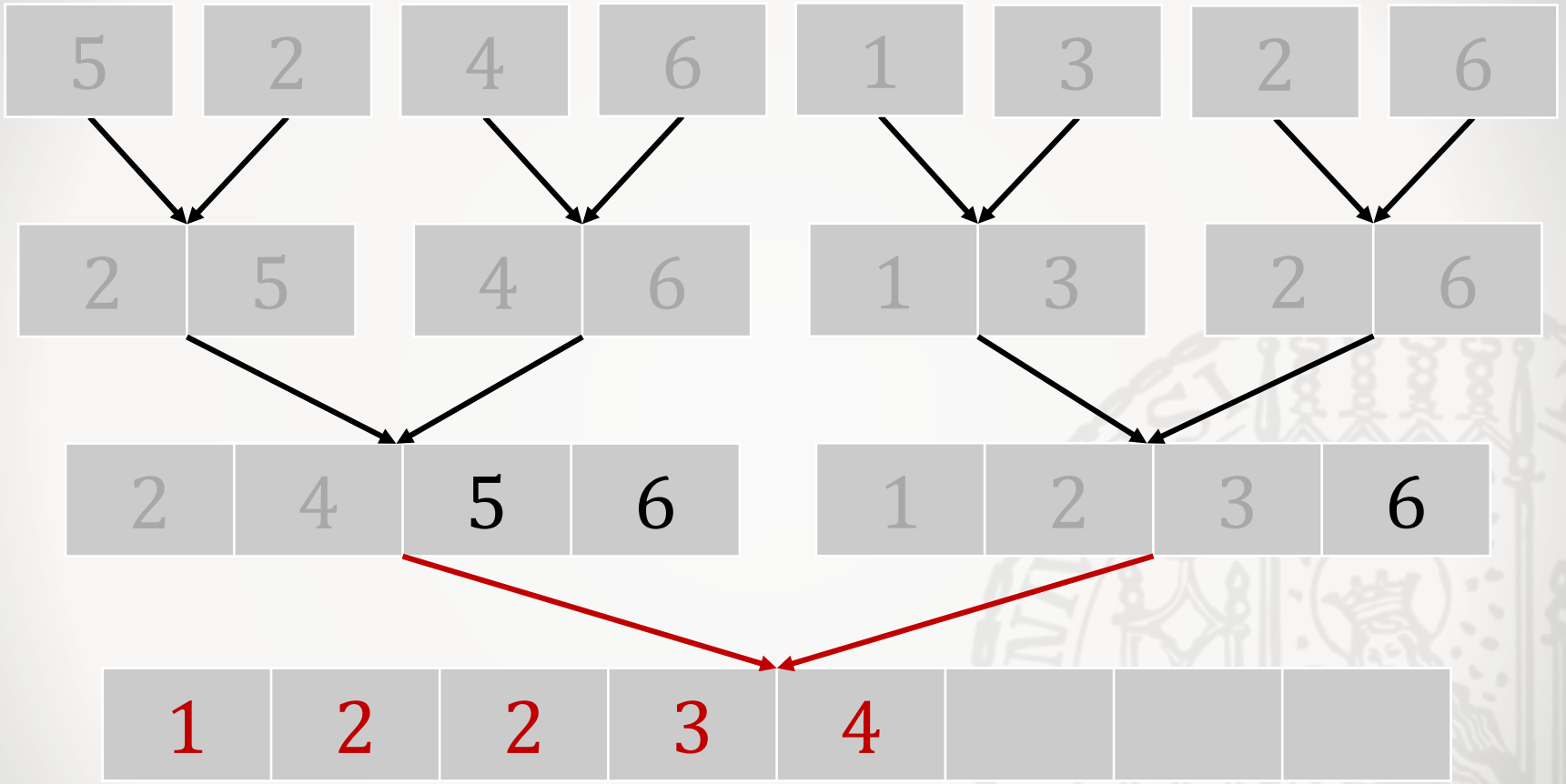
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?\ \ 2 < 2?\ \ 4 < 2?\ \ 4 < 3?$

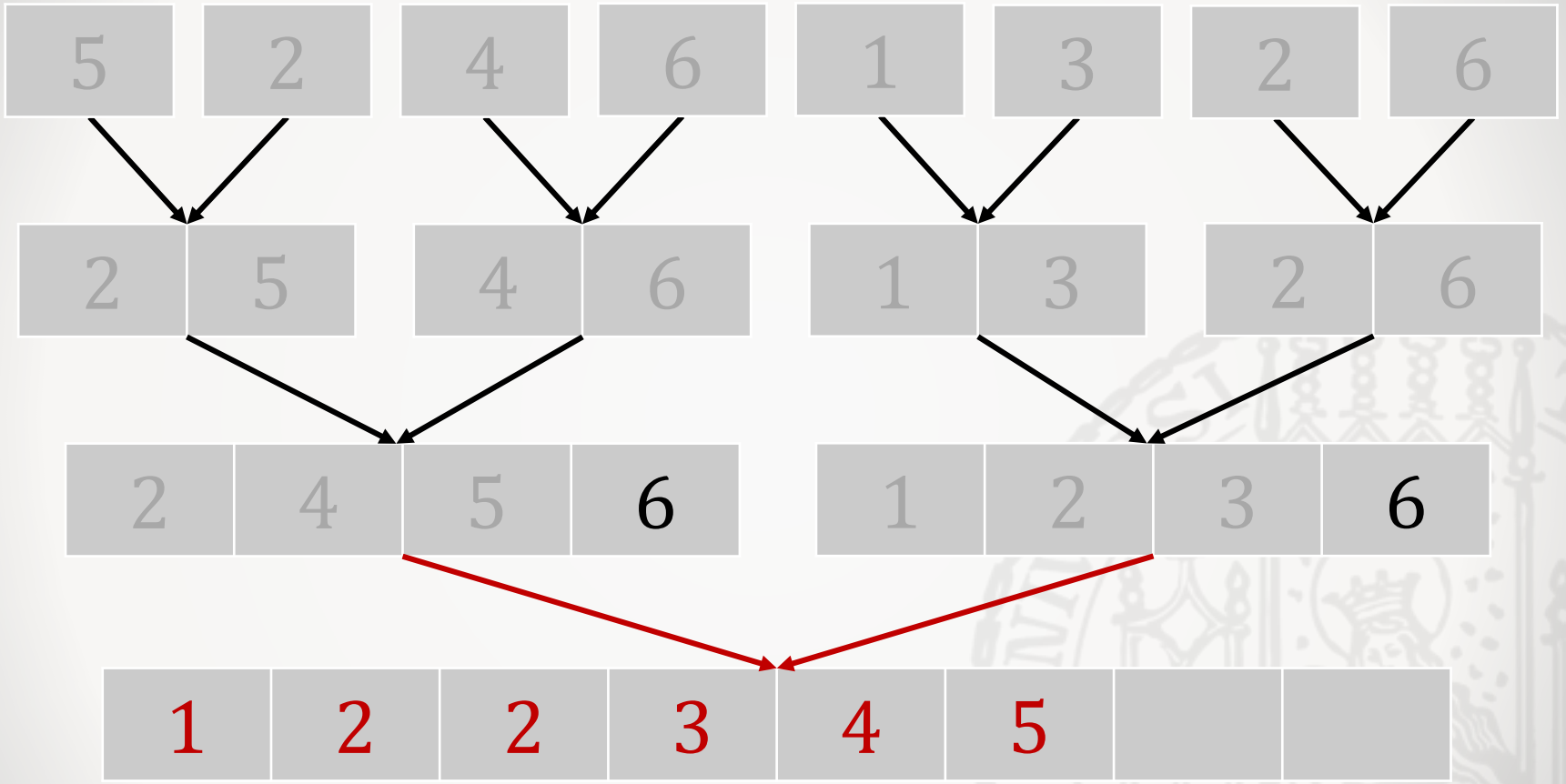
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?\ 2 < 2?\ 4 < 2?\ 4 < 3?\ 4 < 6?$

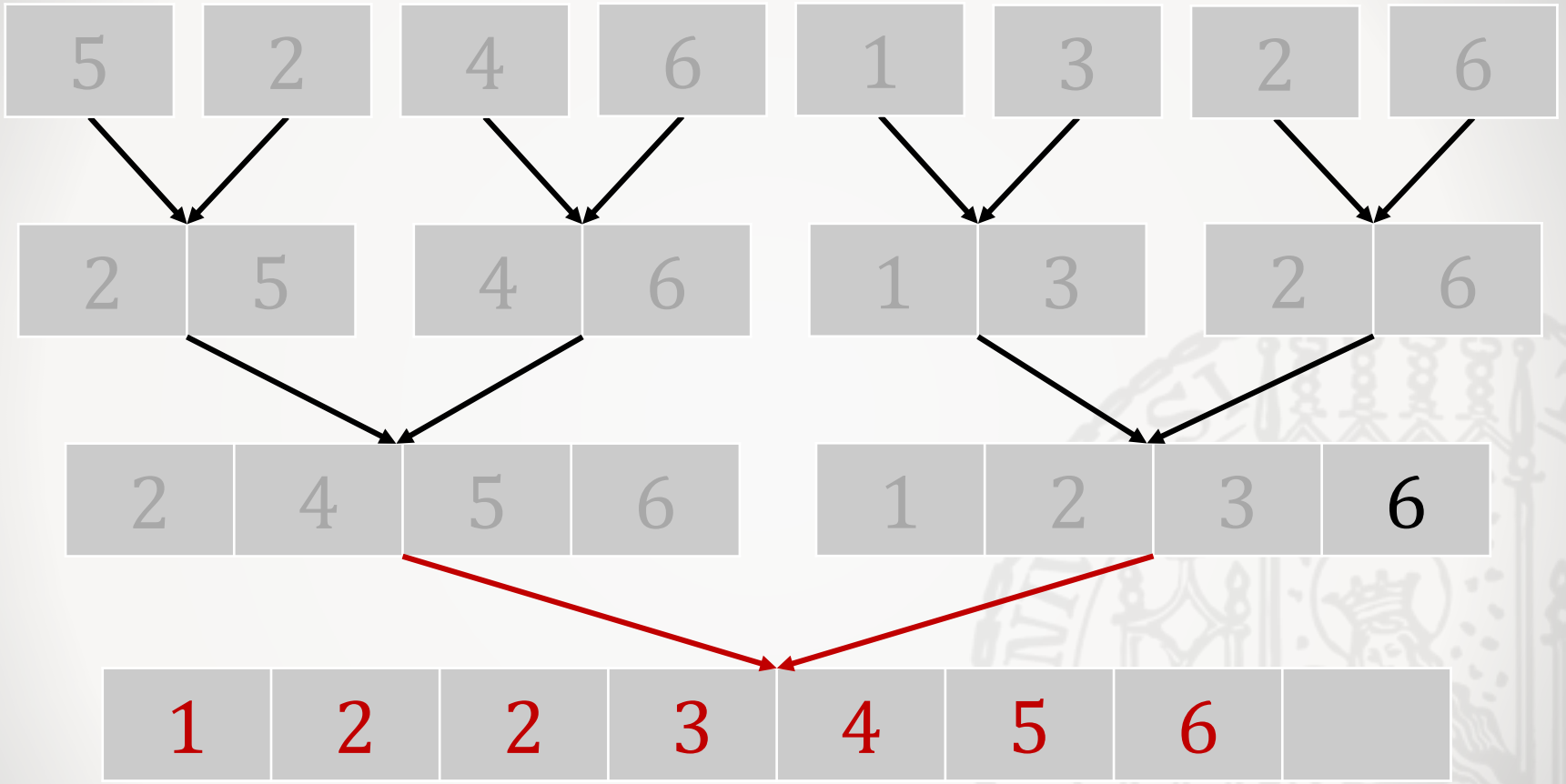
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?\ 2 < 2?\ 4 < 2?\ 4 < 3?\ 4 < 6?\ 5 < 6?$

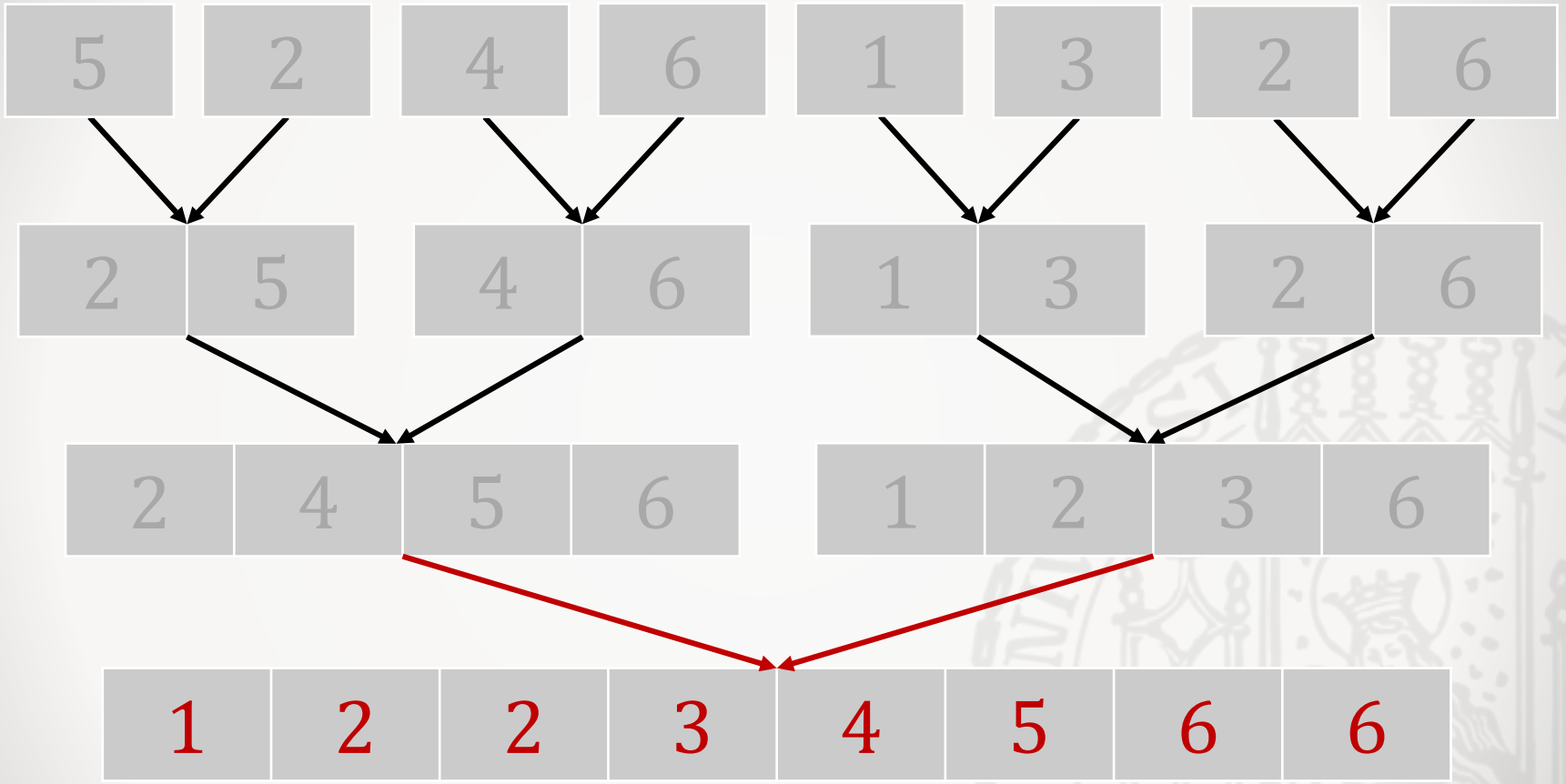
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?\ 2 < 2?\ 4 < 2?\ 4 < 3?\ 4 < 6?\ 5 < 6?\ 6 < 6?$

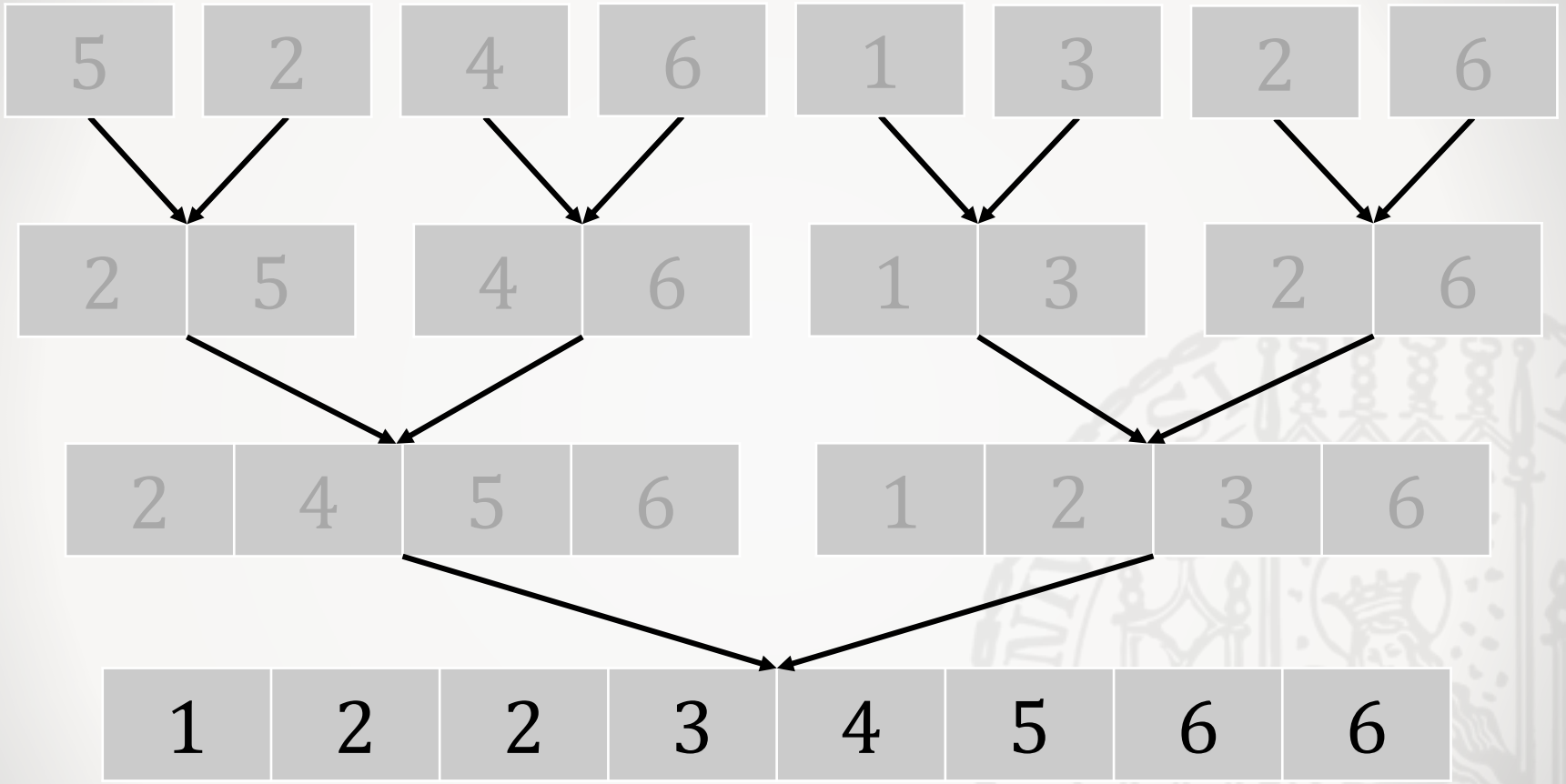
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?\ 2 < 2?\ 4 < 2?\ 4 < 3?\ 4 < 6?\ 5 < 6?\ 6 < 6?$

MergeSort: Merge



Vergleiche: $4 + 6 + 7 = 17$ (InsertionSort: 18)

$2 < 1?\ 2 < 2?\ 4 < 2?\ 4 < 3?\ 4 < 6?\ 5 < 6?\ 6 < 6? \Rightarrow$ 7 Vergleiche

MergeSort: Implementierung

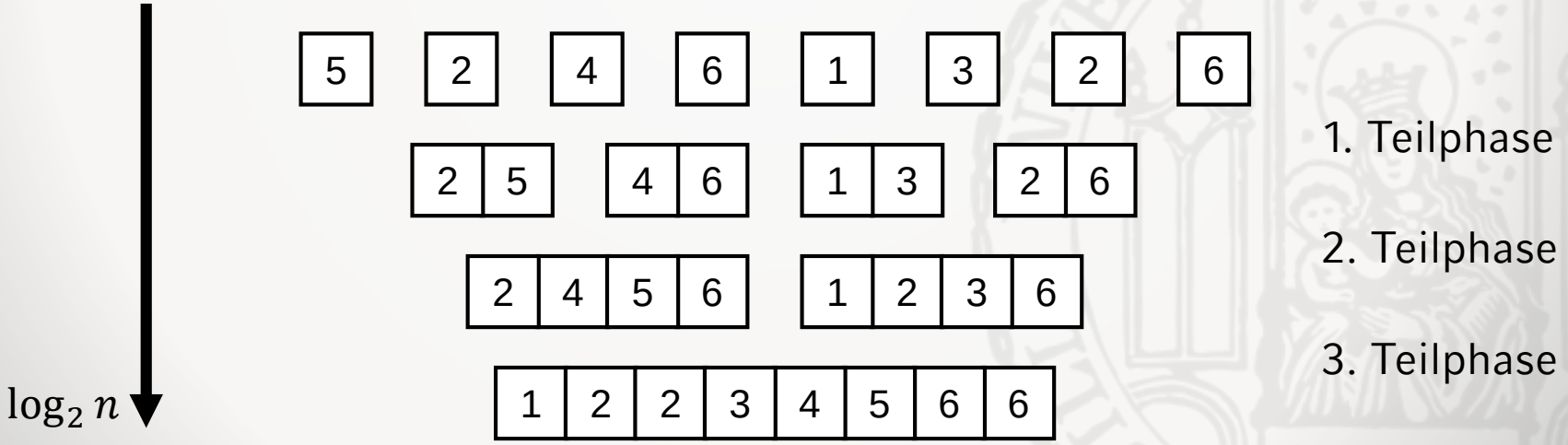
```
def mergeSort(array):
  if len(array) > 1:
    mid = len(array) // 2
    lefthalf = array[:mid]
    righthalf = array[mid:]

    mergeSort(lefthalf)
    mergeSort(righthalf)
    merge(array, lefthalf, righthalf)
```

```
def merge(array, lefthalf, righthalf):
  i, j, k = 0, 0, 0
  while i < len(lefthalf) and j < len(righthalf):
    if lefthalf[i] <= righthalf[j]:
      array[k] = lefthalf[i]
      i=i+1
    else:
      array[k] = righthalf[j]
      j=j+1
    k=k+1

  while i < len(lefthalf):
    array[k] = lefthalf[i]
    i=i+1
    k=k+1

  while j < len(righthalf):
    array[k] = righthalf[j]
    j=j+1
    k=k+1
```

MergeSort: Komplexitätsanalyse

Erinnerung:
Binärbaum mit n Knoten hat maximal Höhe $h = \log_2 n$.

- Wieviele Vergleiche werden benötigt:
  - In der Divide-Phase wird nichts verglichen
  - In der Merge-Phase werden maximal $\log_2 n$ Teilphasen (Baumebenen im Beispiel) durchgeführt.
  - Die i-te Teilphase besteht maximal aus $\frac{n}{2^{i+1}}$ Mergeoperationen
  - In jeder Mergeoperation kann es zu $2^i - 1$ Vergleichen kommen (Bei „Verzahnung“, z.B.: [1,3,5,7] [2,4,6,8])



QuickSort

- Idee:
  - Wähle ein Element p der Folge (Pivotelement) aus.
  - Zerlege die Folge in zwei Teilfolgen:
    - Teilfolge 1: Enthält alle Elemente $x \leq p$
    - Teilfolge 2: Enthält alle Elemente $x > p$
  - Sortiere rekursiv auf den Teillisten mit Quicksort.
  - Einelementige Listen sind schon sortiert.
  - Im Gegensatz zu Mergesort:
    - Aufwand liegt im Divide-Schritt.
    - Merge-Schritt ist lediglich Konkatenation.

QuickSort: Beispiel

Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

7	2	4	6	1	3	2	5

QuickSort: Beispiel

Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

7	2	4	6	1	3	2	5

2	4	1	3	2	5	7	6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

7	2	4	6	1	3	2	5

2	4	1	3	2	5	7	6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

QuickSort: Beispiel

Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

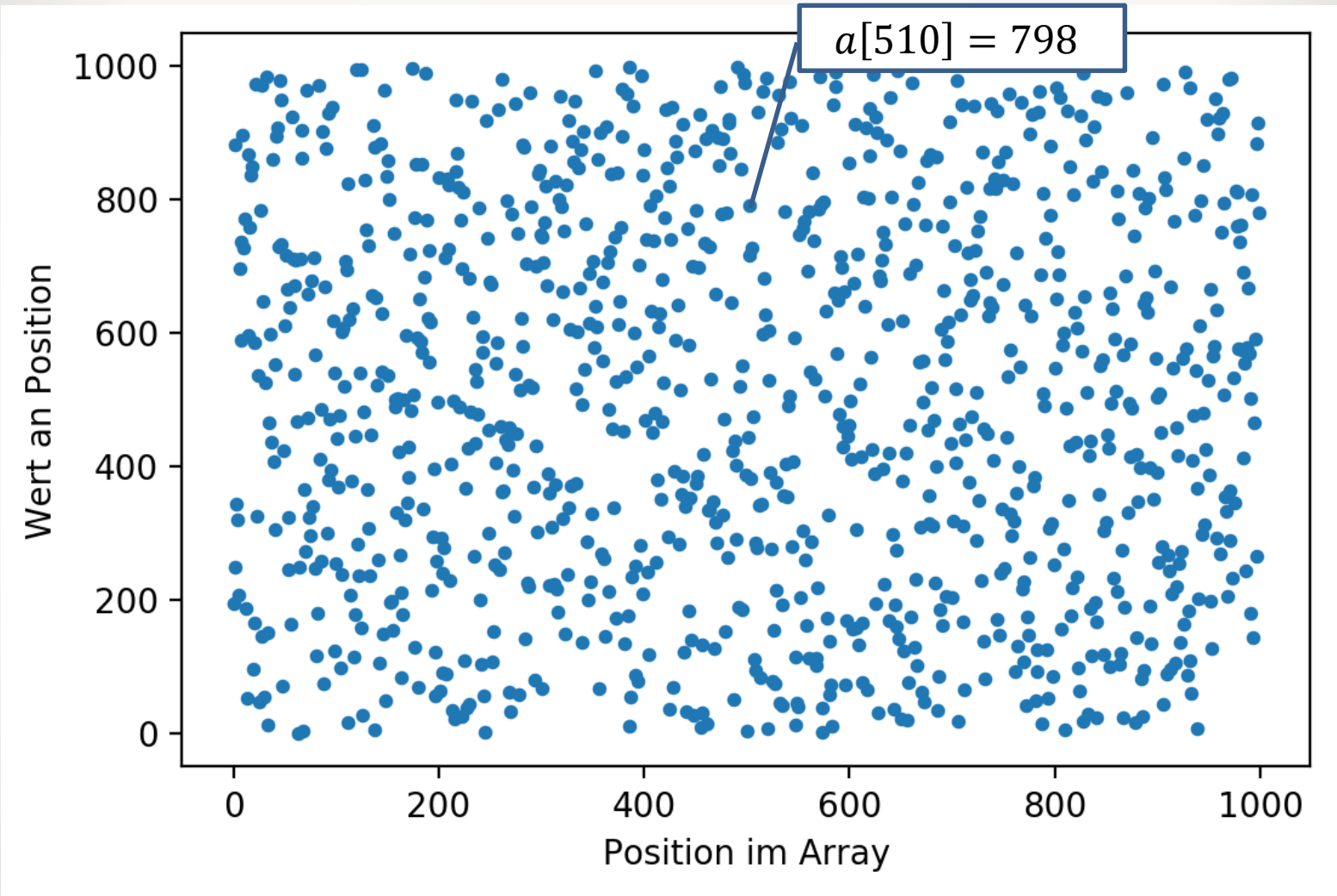
QuickSort: Beispiel

Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

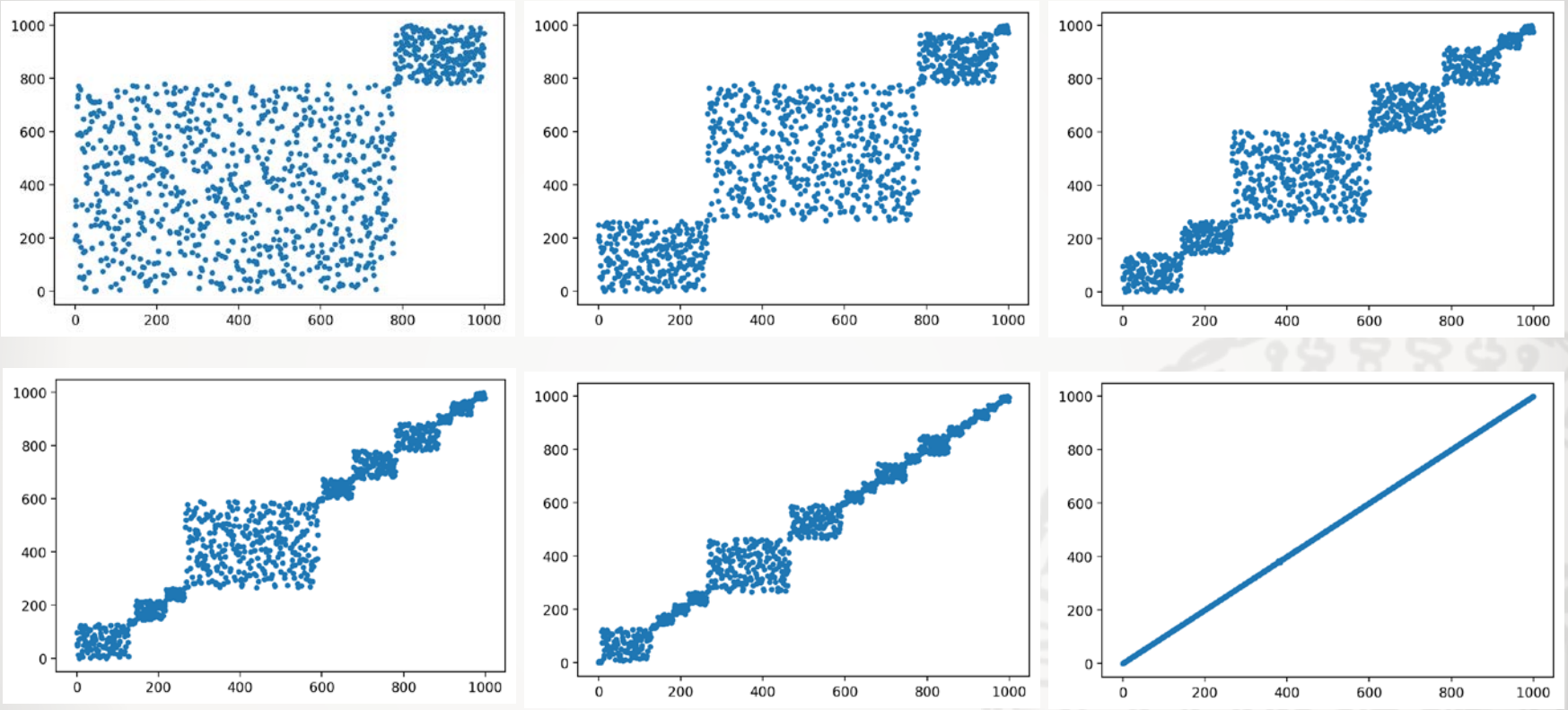
QuickSort: Beispiel

Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

QuickSort: Visualisierung



QuickSort: Visualisierung



QuickSort: Implementierung

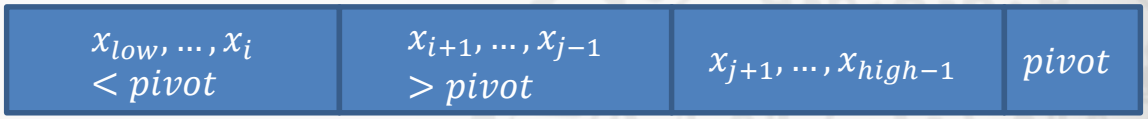
Pivot:
Wir benutzen hier das letzte Listenelement als Pivot!

```
def quickSort(arr, low, high):
  if low < high:
    pi = partition(arr, low, high)
    quickSort(arr, low, pi-1)
    quickSort(arr, pi+1, high)
```

← Position des Pivots

← } Rekursionen

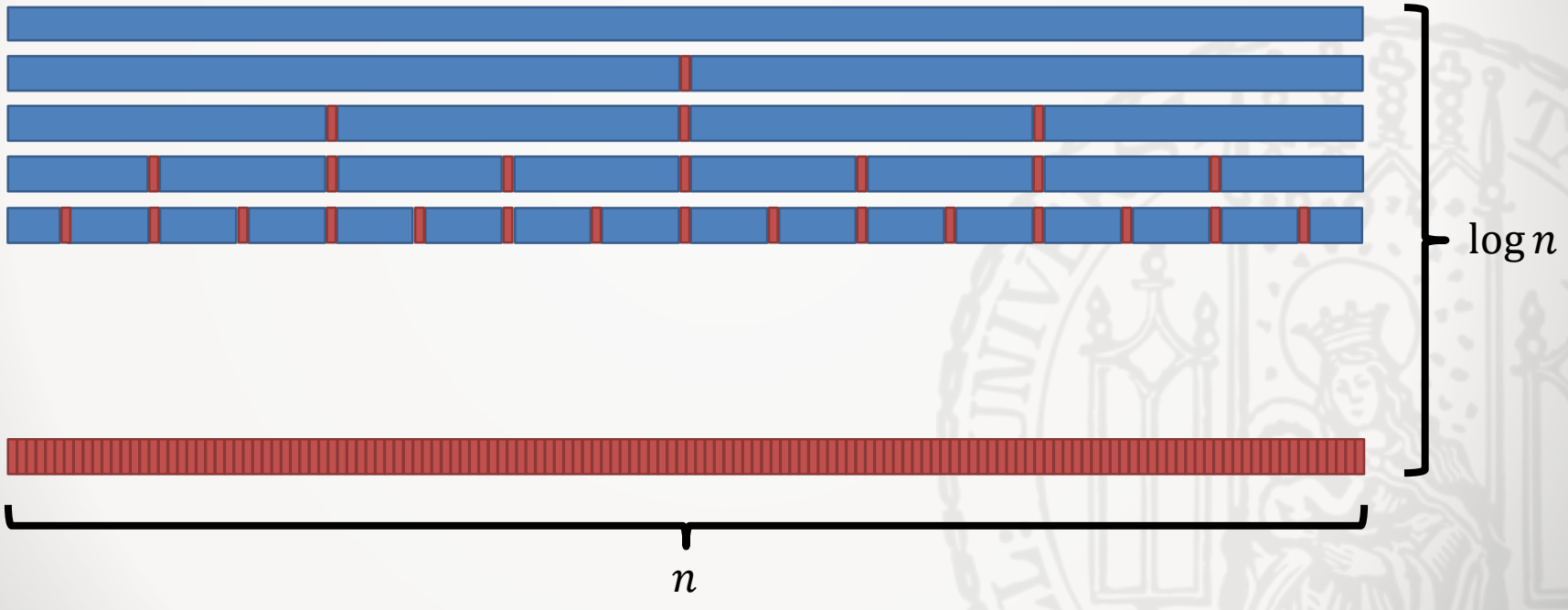
```
def partition(arr, low, high):
  i = (low-1) ← Pivot
  pivot = arr[high]

  for j in range(low, high):
    if arr[j] <= pivot:
      i += 1
      arr[i], arr[j] = arr[j], arr[i]
  arr[i+1], arr[high] = arr[high], arr[i+1]
  return(i+1)
```



Pivot an die richtige Stelle tauschen

QuickSort: Komplexitätsanalyse Best-Case

- Wir betrachten die Anzahl der Vergleichsoperationen.
- Bevor wir das Mastertheorem formell anwenden, erst grafisch.
- Best-Case: Die Folge zerfällt immer in zwei gleich große Teile.



QuickSort: Komplexitätsanalyse Best-Case

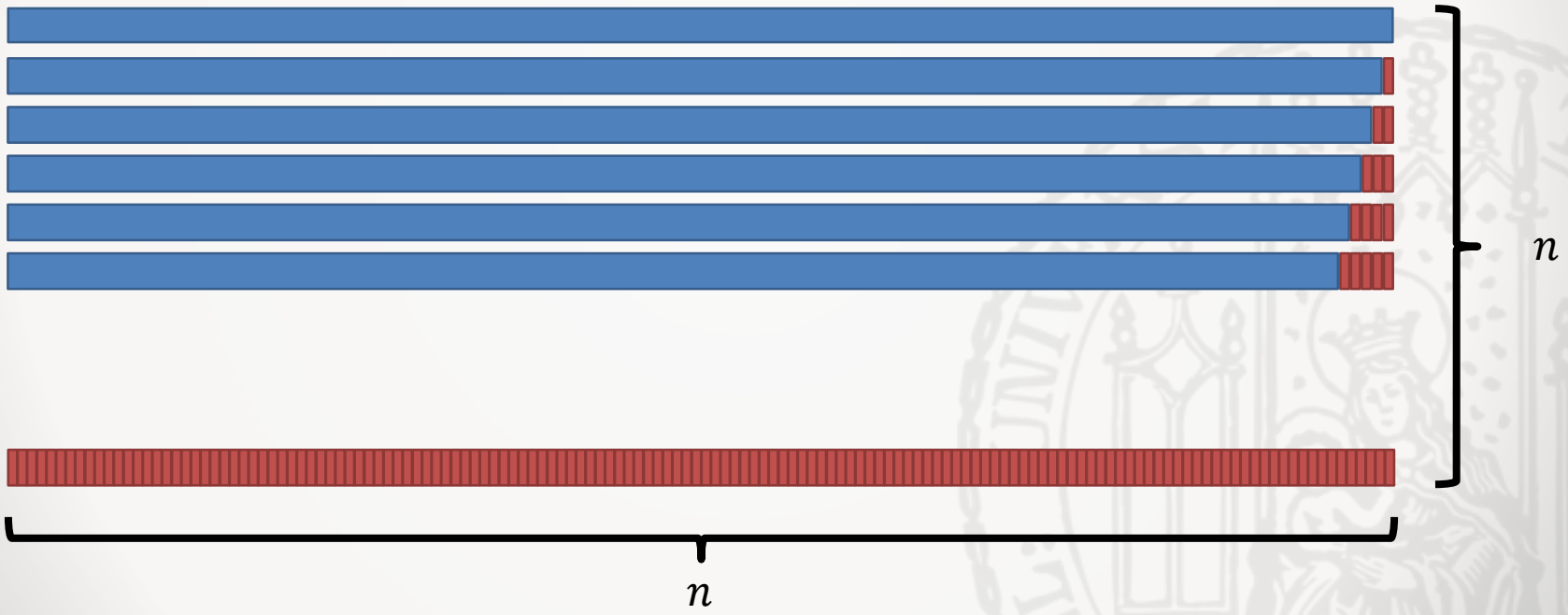
- Das Pivot wird mit allen übrigen Elementen verglichen.
- Die beiden Teilfolgen enthalten nicht mehr das Pivot.
- Beide Teilfolgen werden ebenfalls sortiert.
- Wenn eine Folge nur noch aus einem Element besteht, werden keine Vergleiche mehr benötigt.
- Für die Komplexität im Best Case ergibt sich:

$$T(n) = \begin{cases} 2 \cdot T\left(\frac{n-1}{2}\right) + n - 1 & , n > 1 \\ 0 & , n = 1 \end{cases}$$

- Mit Mastertheorem gilt $T(n) \in O(n \log n)$.

QuickSort: Komplexitätsanalyse Worst-Case

- Die Folge zerfällt schlimmstenfalls immer in
  - eine leere Folge
  - und eine Folge der Länge $n - 1$.



QuickSort: Komplexitätsanalyse Worst-Case

- Der Aufrufbaum ist somit ein entarteter Baum (ein linearer Zweig)

$$T(n) = \begin{cases} T(n-1) + n - 1 & , n > 1 \\ 0 & , n = 1 \end{cases}$$

Mit Mastertheorem gilt $T(n) \in O(n^2)$.

- Im schlimmsten Fall ist QuickSort damit genauso langsam wie BubbleSort / SelectionSort / InsertionSort

QuickSort: Komplexitätsanalyse Average-Case (1/2)

- Annahmen:
  - Alle n Schlüssel seien verschieden.
  - Die Wahrscheinlichkeit der Eingabeanordnung sei $\frac{1}{n!}$.
  - Jede der $n!$ Permutationen ist damit gleich wahrscheinlich.
- Bei der Partitionierung entstehen zwei Teilfolgen, die Längen im Interval $(0, n-1)$ annehmen können. Jede Länge ist gleich wahrscheinlich. Der Möglichkeitsraum der Längenpaare ist daher

$$\{(0, n-1), (1, n-2), \dots, (n-2, 1), (n-1, 0)\}$$

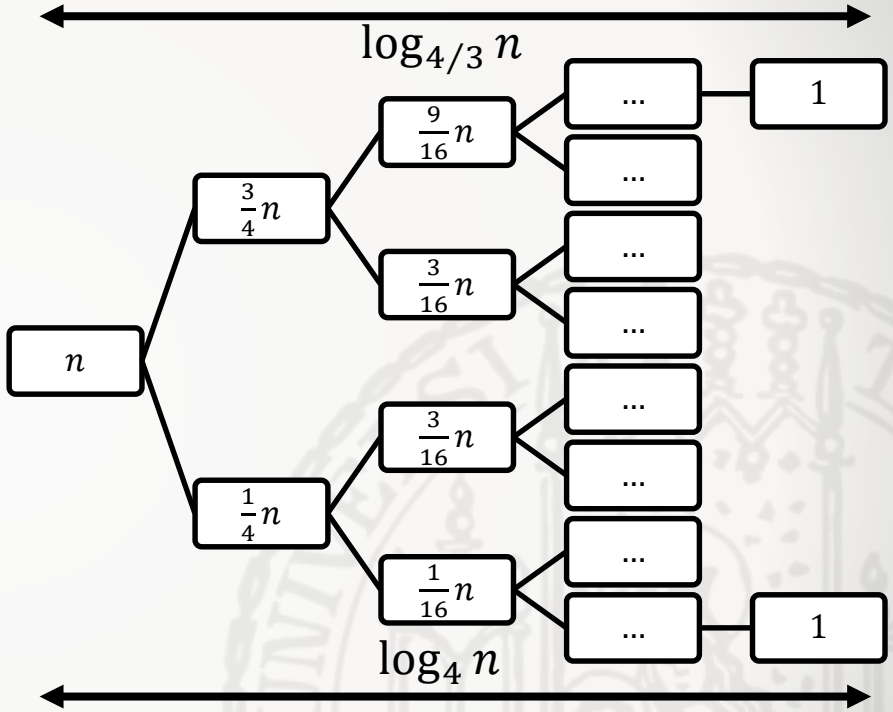
- Im Durchschnitt ist daher die Länge der größeren Teilfolge:

$$L_1 = \frac{2}{n} \sum_{i=\frac{n}{2}}^{n-1} i = \frac{3}{4} n - \frac{3}{2}$$

- Wir können die Länge zu $L_1 = \frac{3}{4} n$ abschätzen, da es dadurch nur langsamer wird.

QuickSort: Komplexitätsanalyse Average-Case (2/2)

- $L_1 = \frac{3}{4}n, L_2 = \frac{1}{4}n$
- Der längste Rekursionspfad besteht aus $\log_{4/3} n$ Knoten.
- Jedes Level des Aufrufbaums nutzt $O(n)$ Vergleiche
- Damit ergeben sich $O(n \log n)$ Vergleiche im durchschnittlichen Fall.



QuickSort: Pivotelement

- Pivot (frz.): Dreh-/Angelpunkt
- Die Wahl des Pivots beeinflusst stark die Ausführung und Rekursionstiefe des Algorithmus.
  - Beispiel: Sortiere 7-elem. Liste und nutze 1. Element als Pivot.

4	2	1	3	6	7	5
2	1	3	4	6	7	5
1	2	3	4	5	6	7
1	2	3	4	5	6	7

1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

- Wie Pivot wählen, damit Worst-Case vermieden wird?

QuickSort: Pivotelement einfache Auswahlstrategien

- „Wähle immer erstes Element“ $p = x_1$
  - Einfach zu implementieren
  - (Teilweise) vorsortierte Listen führen zu schlechter Laufzeit
- „Wähle immer letztes Element“ $p = x_n$
  - Gleiche Eigenschaften wie „ Wähle immer erstes Element“
- „Wähle immer mittleres Element“ $p = x_{\lceil n/2 \rceil}$
  - Ebenfalls einfach zu implementieren
  - (Teilweise) vorsortierte Listen nicht kritisch
  - Auch hier lassen sich Worst-Case-Szenarien erstellen
    - Sortiere 6421357

6	4	2	1	3	5	7
1	6	4	2	3	5	7
1	2	6	4	3	5	7
1	2	3	6	4	5	7
1	2	3	4	6	5	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

QuickSort: Pivotelement Median

- Naiver Algorithmus $median$
  - Gegeben ist eine Folge $X = x_1 x_2 \dots x_n$
  - Sortiere X
  - Gib Element an Position $\lceil n/2 \rceil$ aus

- Median ist immer optimal bzgl. Partitionierungsgröße.
  Aber:
  - Best Case von QuickSort benötigt $O(n \log n)$
  - Wegen Median: In jedem Schritt Sortieren mit $O(n \log n)$
  - Damit ist Sortierung aber schon anderweitig gelöst

- Geht das besser?

QuickSort: Pivotelement Median-Of-Three

- Deterministische Variante:
  - Wähle drei Elemente, meistens: $x_1, x_{\lceil n/2 \rceil}, x_n$
  - Pivot ist dann $\text{median}\left(x_1, x_{\lceil n/2 \rceil}, x_n\right)$

- Auch hier kann ein Worst-Case konstruiert werden.
  - Wie sieht dieser aus?

- Sortieren der drei selektierten Werte sinnvoll vor dem Teilen.
  - Betrachte z.B. $(n, 1, 2, 3, 4, 5, \ldots, n-2, n-1, 0)$

- Kann erweitert werden zu *Median-of-k* für $k > 3$ viele Elemente.
  - Mehraufwand macht sich jedoch nicht bezahlt.

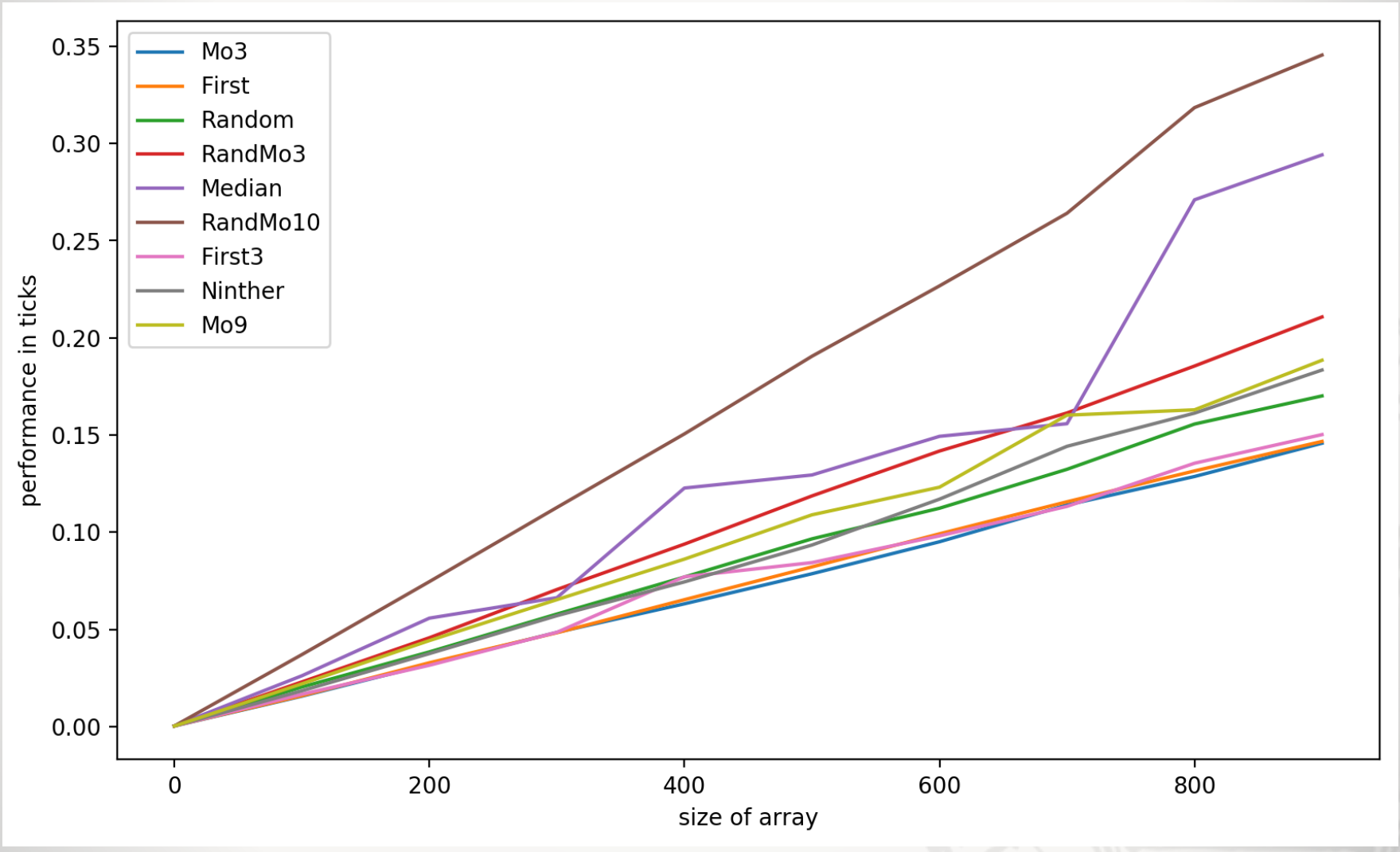
QuickSort: Pivotelement Randomized

- Bei deterministischer Wahl eines bestimmten Elements lässt sich stets ein Worst-Case konstruieren.

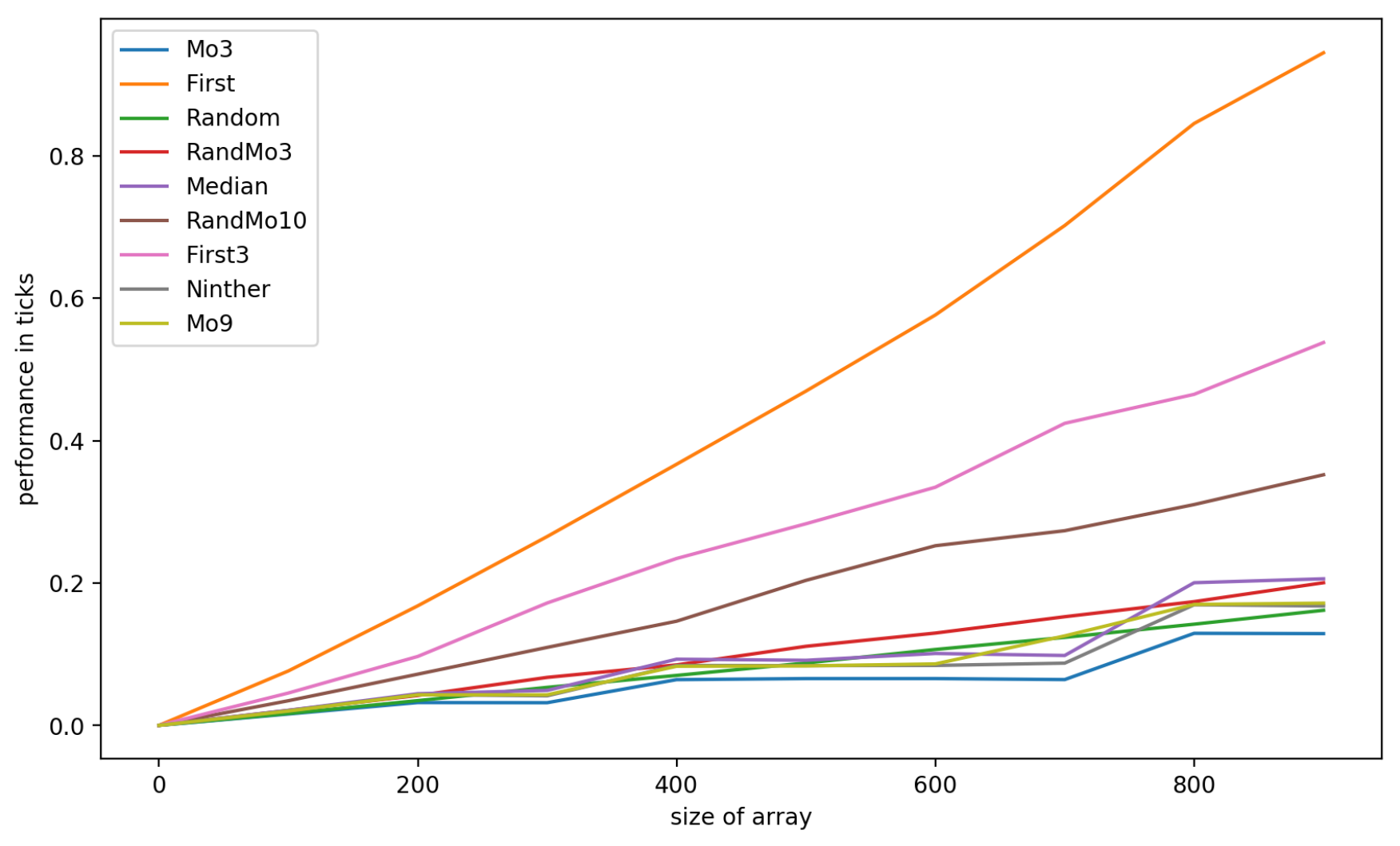
- Lösung: Nicht-deterministisches QuickSort
  - Wähle zufälliges Element als Pivot
  - In der Praxis weit verbreitet (einfach und schnell)
  - Wahrscheinlichkeit für Worst-Case sehr klein

- Nachteil: Pseudo Random Number Generator ist nicht umsonst.
  - Pivotelement auswählen ist relativ atomare Operation
  - Frequenz des Aufrufs des PRNG hoch
  - Auf diesem Operationslevel ist der Unterschied messbar.

QuickSort: Vergleich auf gleichverteilt-gemischte Listen



QuickSort: Vergleich auf totalsortierter Liste



QuickSort: Pivotelement Fazit

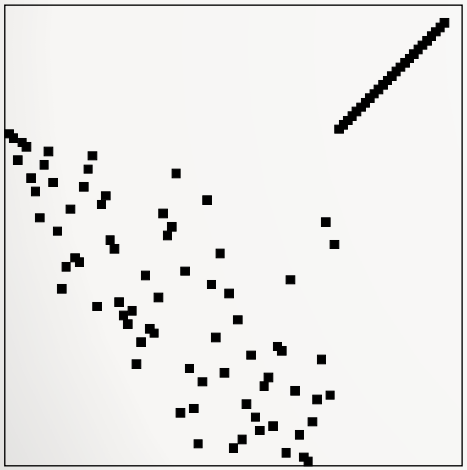
- In der Regel sind einfache Methoden meist eine solide Strategie

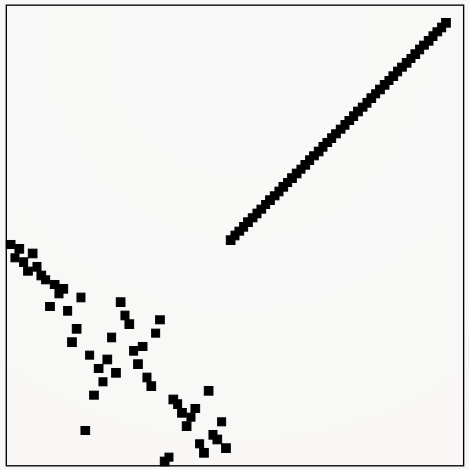
- Zur Auswahl der passenden Strategie:
  - Welche Listen werden erwartet?
    - Vorsortiert, Teilsortiert, Verteilung der Entropie bekannt?
  - Sind Worst-Cases ein relevanter Faktor?
  - Median berechnen auf vielen Werten ineffizient
  - Zufallsgeneratoren sparsam nutzen, wenn Operationen schon häufig genutzt werden

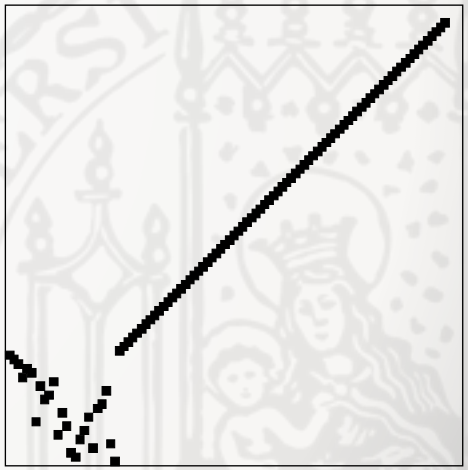
- Insgesamt bleibt QuickSort in $O(n \log n)$ (außer für Median), der Unterschied liegt in der Konstanten c, also $c\, n \log n$

HeapSort

- SelectionSort: Wähle n-mal das jeweils nächste Maximum in $O(n)$
- Falls das immer in $O(\log n)$ ginge, hätten wir ein $O(n \log n)$ Verfahren
- Idee: Stelle das jeweils nächste Maximum in $O(\log n)$ ganz nach vorne.

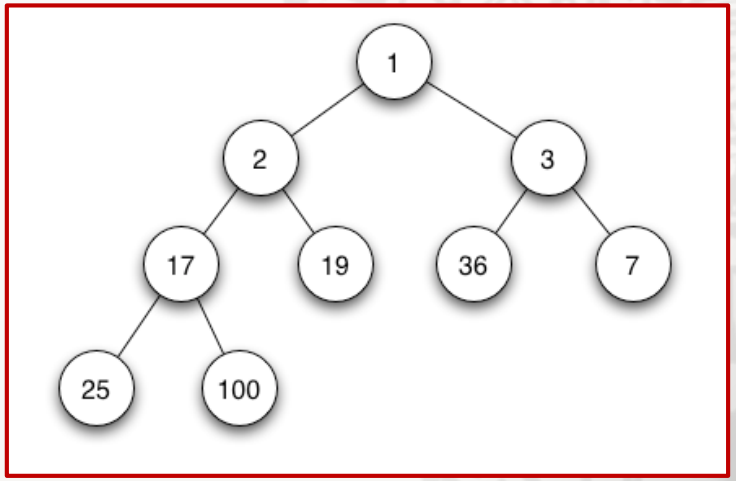






Heap-Struktur

- Ein Heap ist ein Binärbaum, der folgende Eigenschaften erfüllt:
  - Links-vollständig, alle Ebenen sind vollständig gefüllt, nur die letzte kann von rechts her unvollständig sein.
  - Knoten enthalten eine Menge von Schlüsselwerten, auf der eine totale Ordnung definiert ist.
  - Für jeden Knoten gilt je nach Heap-Variante:
    - Max-Heap: Vorgänger haben größere Schlüssel als Nachfolger
    - Min-Heap: Vorgänger haben kleiner Schlüssel als Nachfolger
  - Heaps stellen partiell geordnete Bäume dar.



HeapSort: Ablauf

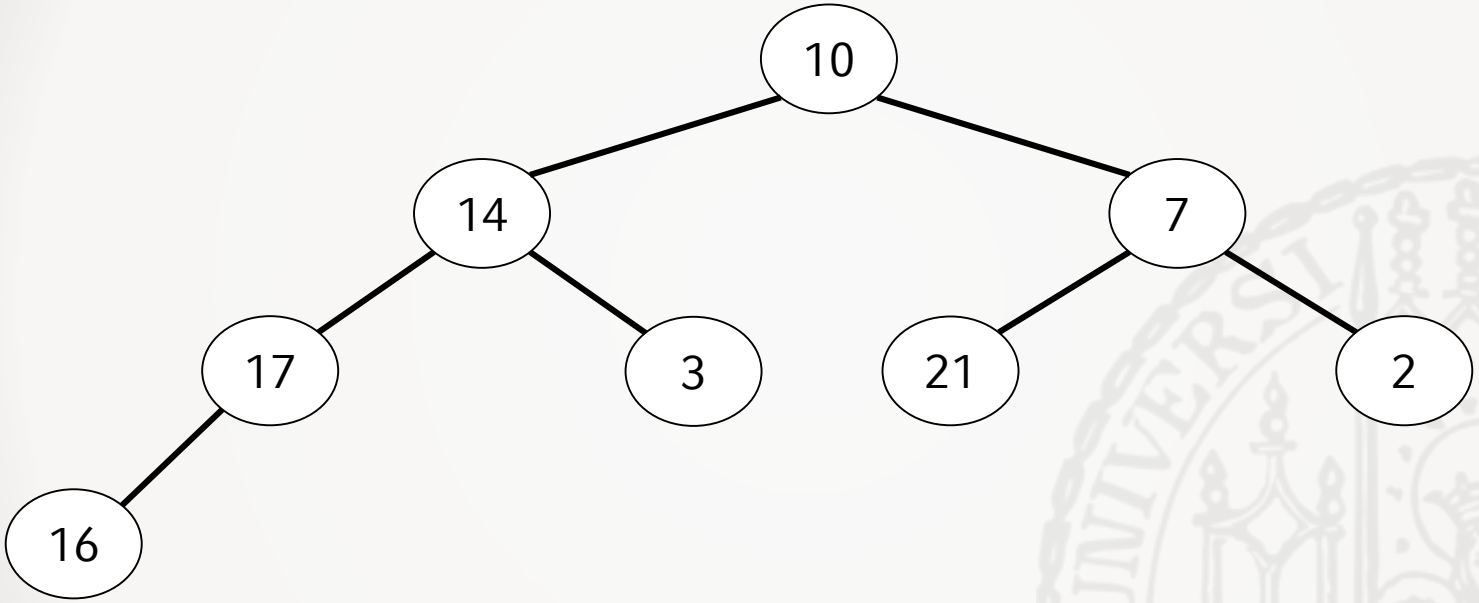
- Füge alle Elemente in einen linksvollständigen Binärbaum ein.
- Heapaufbau: Stelle die Max-Heap-Eigenschaft im Baum her.
  - Von hinten (Blätter) nach vorne (Wurzel):
    Prüfe Heap-Eigenschaft; falls nicht erfüllt, tausche Vorgänger sukzessive mit Nachfolgern, bis Heapeigenschaft erfüllt ist.
- Sortierphase: Für alle Elemente x_i in $(x_0, \ldots, x_{n-1})$:
  - Tausche Wurzel (größtes Element) mit dem $(\mathrm{n} - i)$-ten Element.
  - Stelle für alle Elemente kleiner x_i die Heapeigenschaft wieder her (Versickern)

- Gesamter Algorithmus möglich in-place als Arrayeinbettung.

HeapSort: Beispiel

10	14	7	17	3	21	2	16

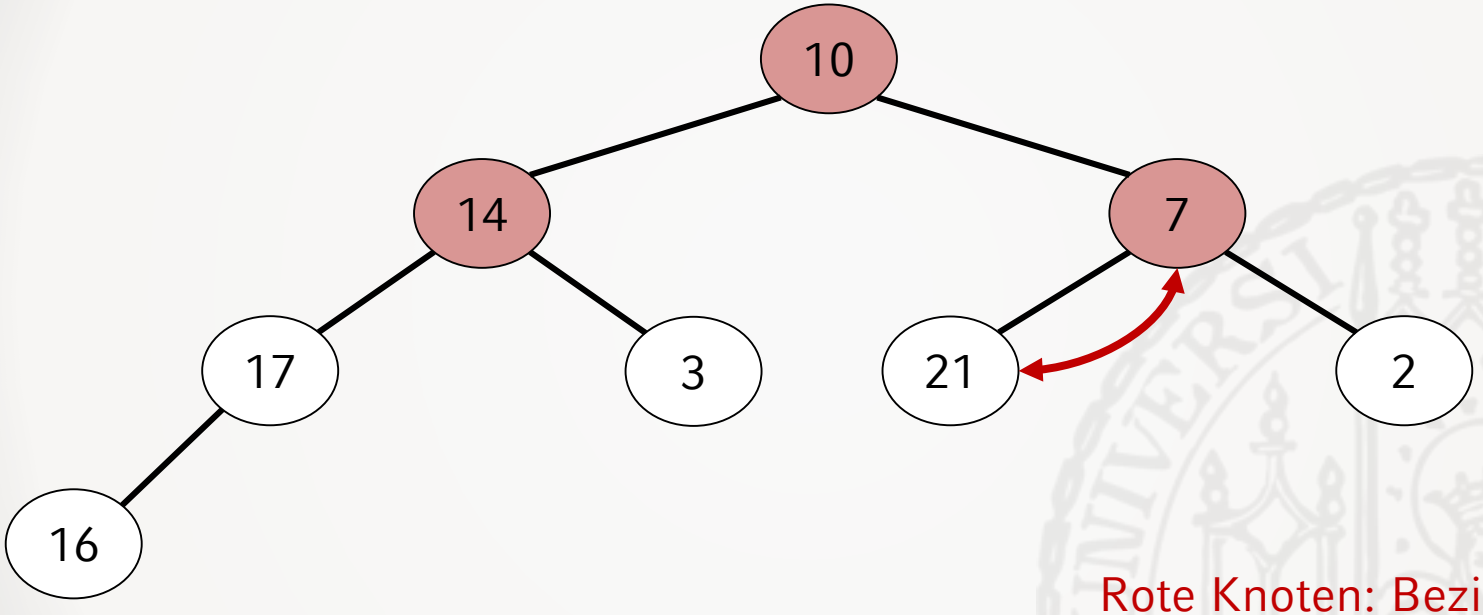
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Einfügen in Binärbaum



HeapSort: Beispiel

10	14	7	17	3	21	2	16

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft herstellen (Heapify)

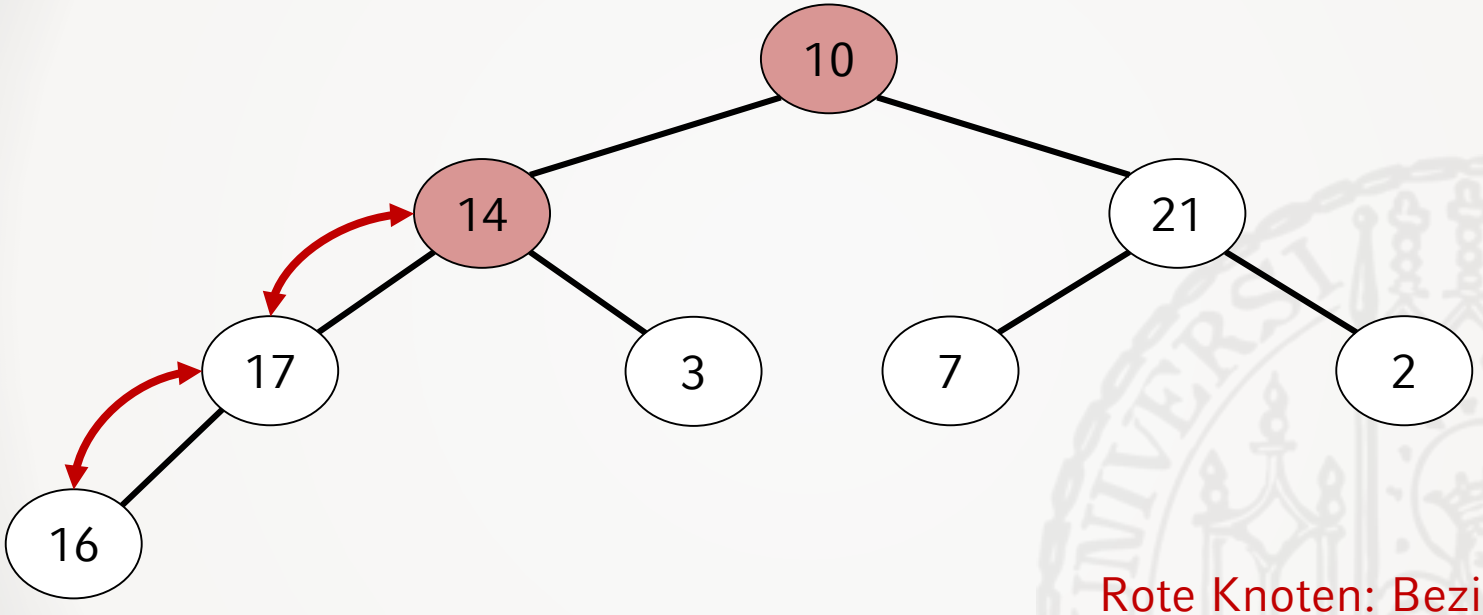


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

10	14	21	17	3	7	2	16

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft herstellen (Heapify)

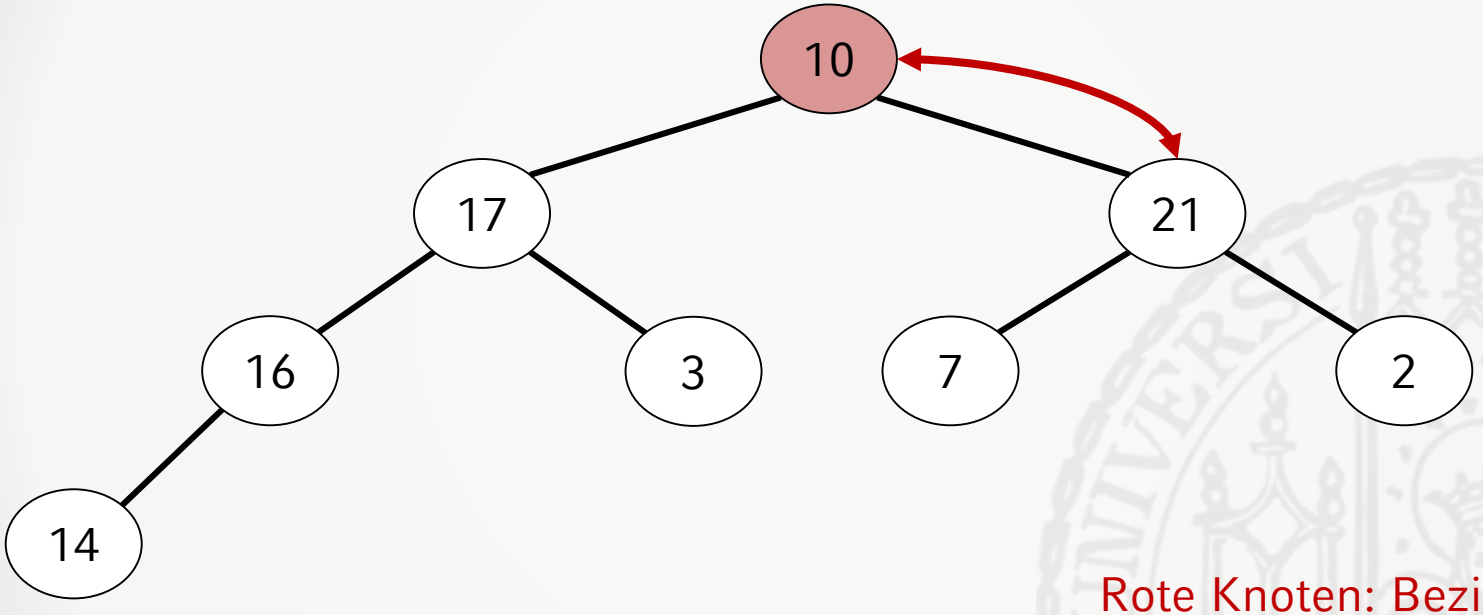


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

10	17	21	16	3	7	2	14

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft herstellen (Heapify)

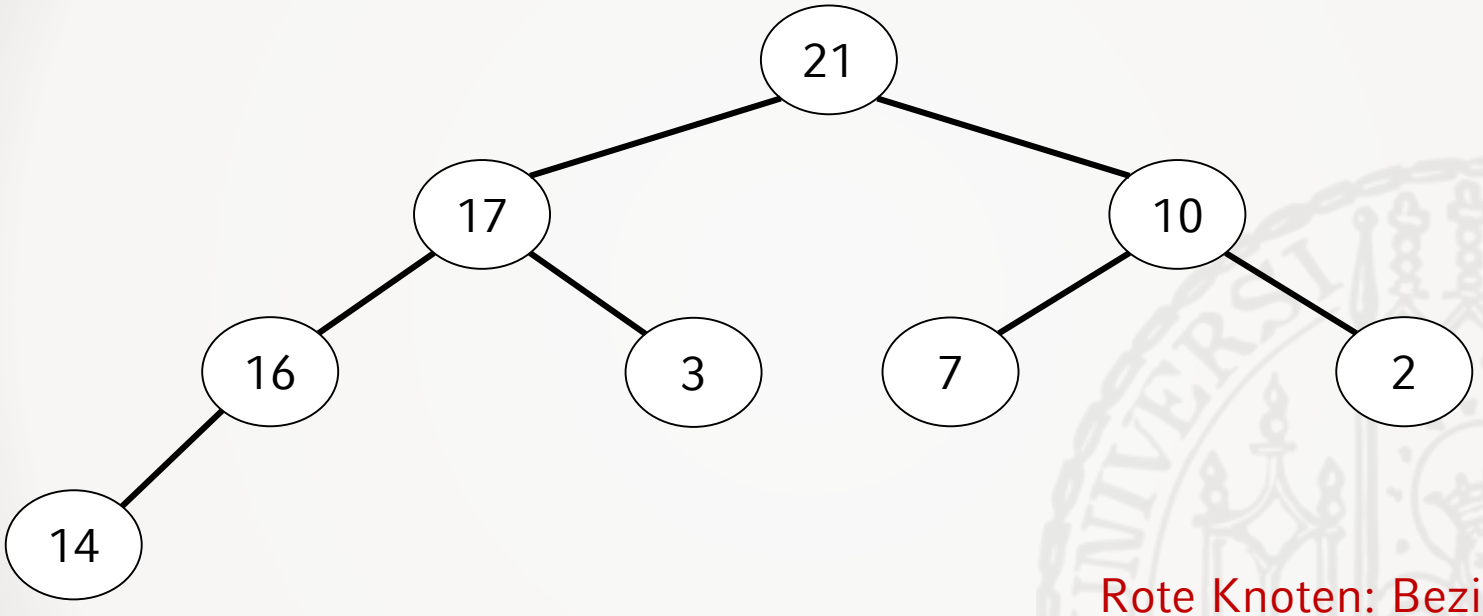


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

21	17	10	16	3	7	2	14

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft hergestellt!

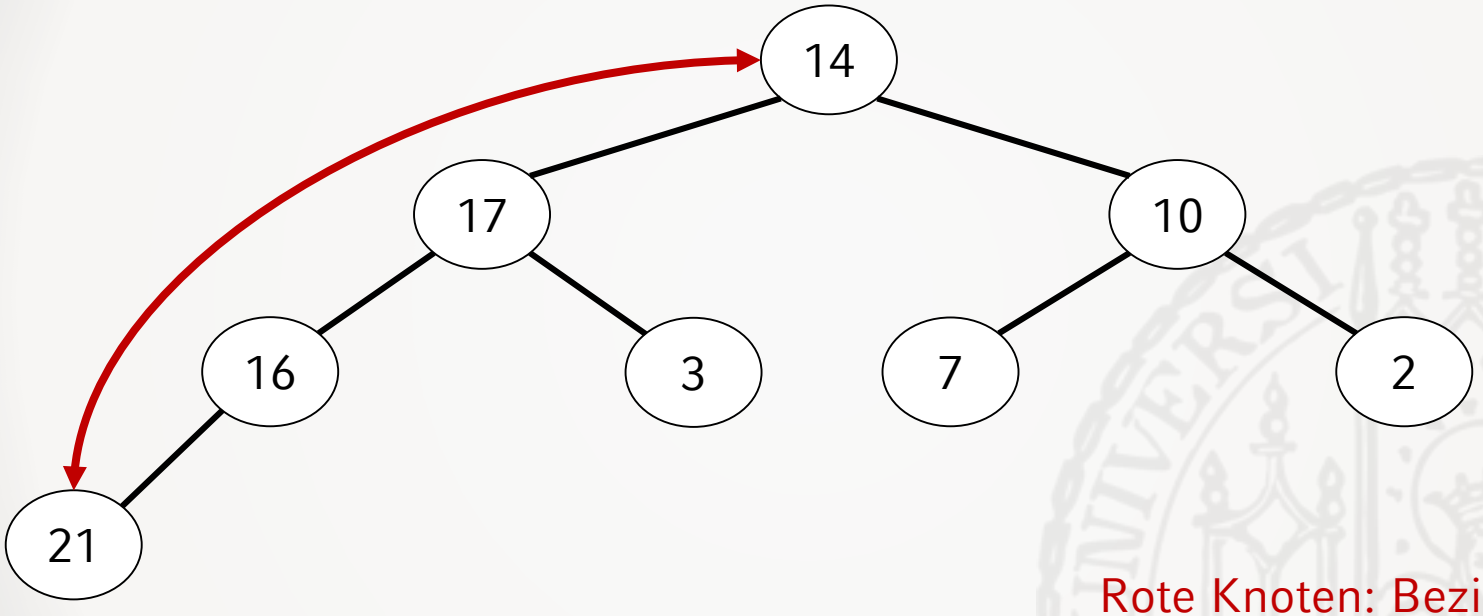


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

14	17	10	16	3	7	2	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element

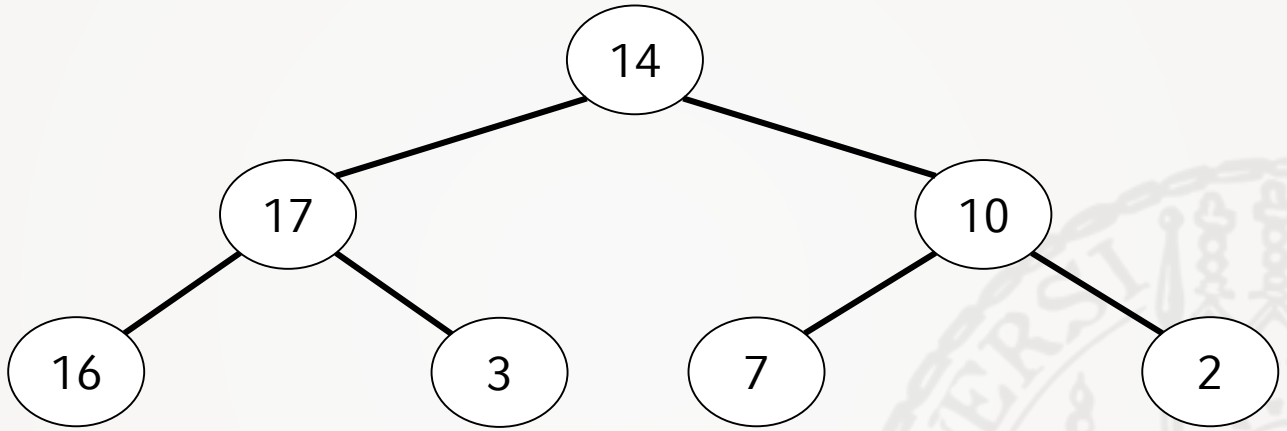


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

14	17	10	16	3	7	2	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur

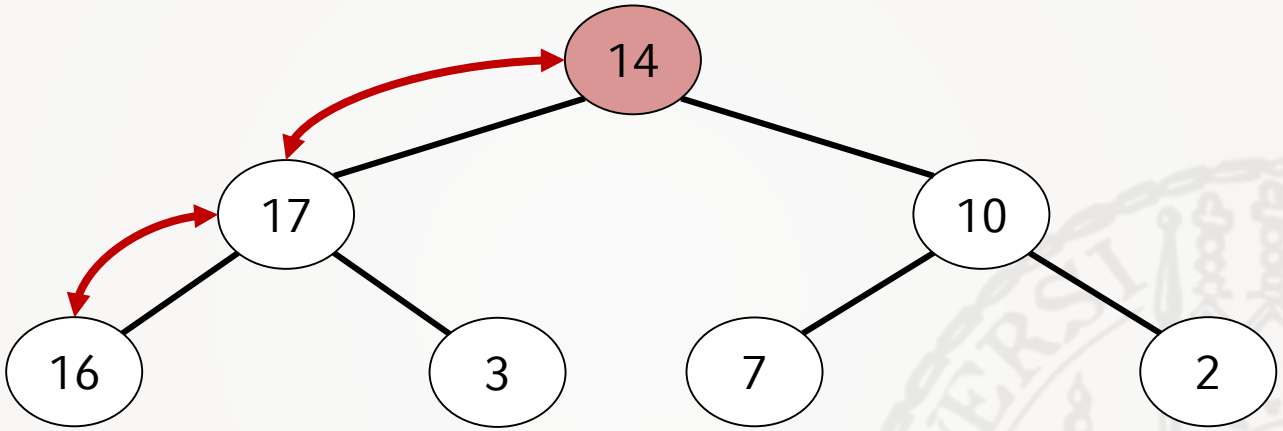


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

14	17	10	16	3	7	2	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

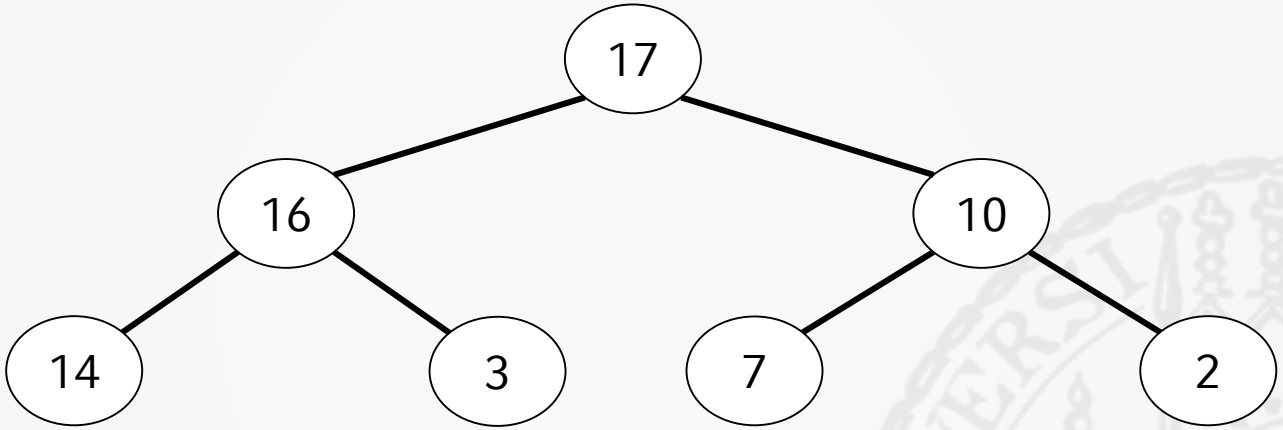


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

17	16	10	14	3	7	2	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

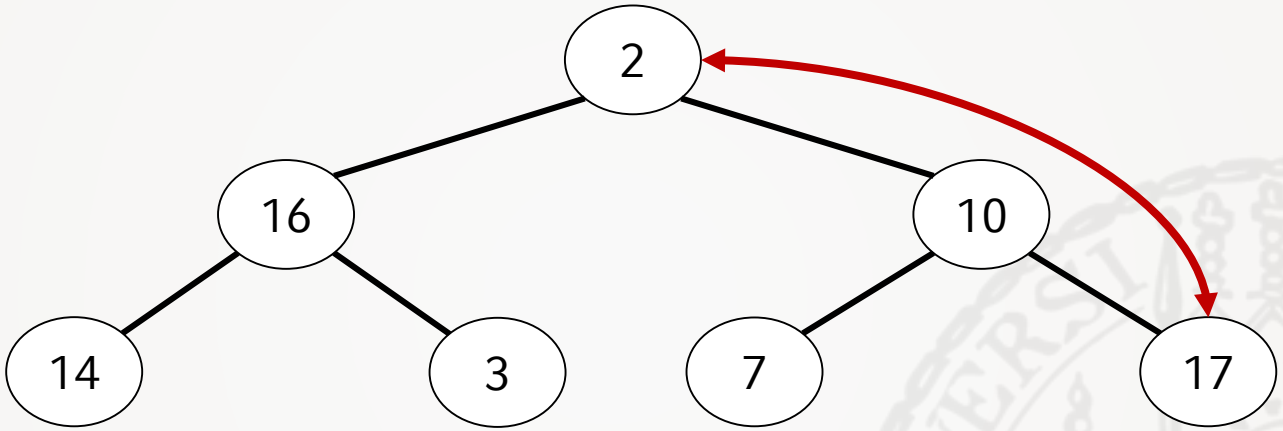


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

2	16	10	14	3	7	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element

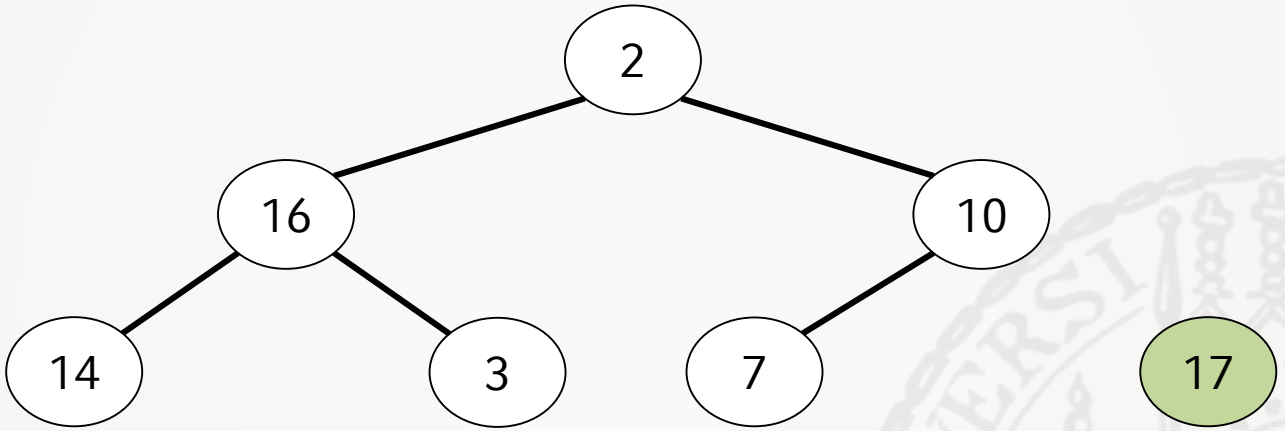


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

2	16	10	14	3	7	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur

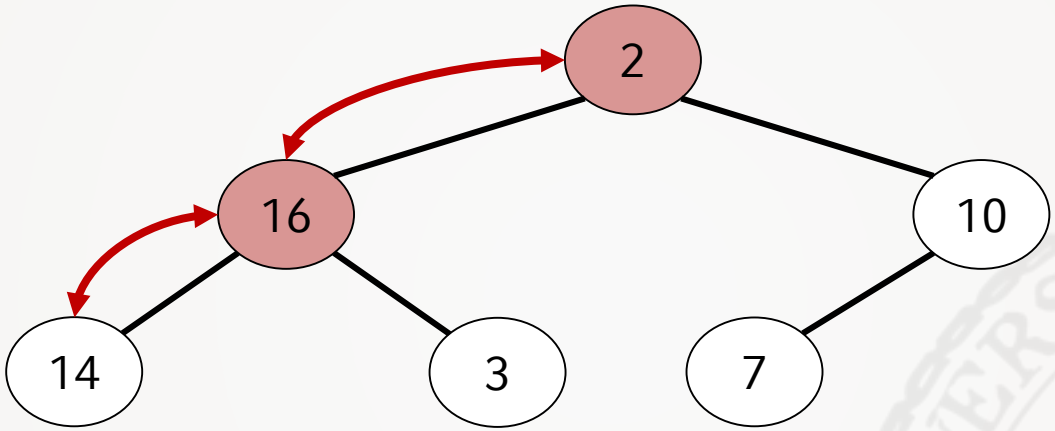


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

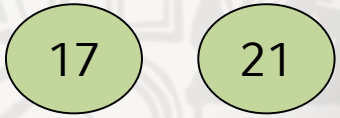
HeapSort: Beispiel

2	16	10	14	3	7	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



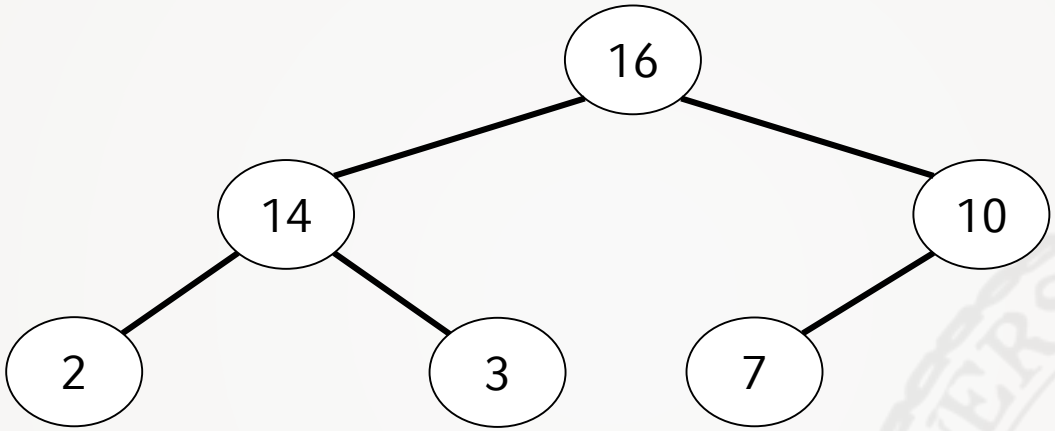
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



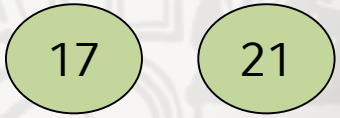
HeapSort: Beispiel

16	14	10	2	3	7	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



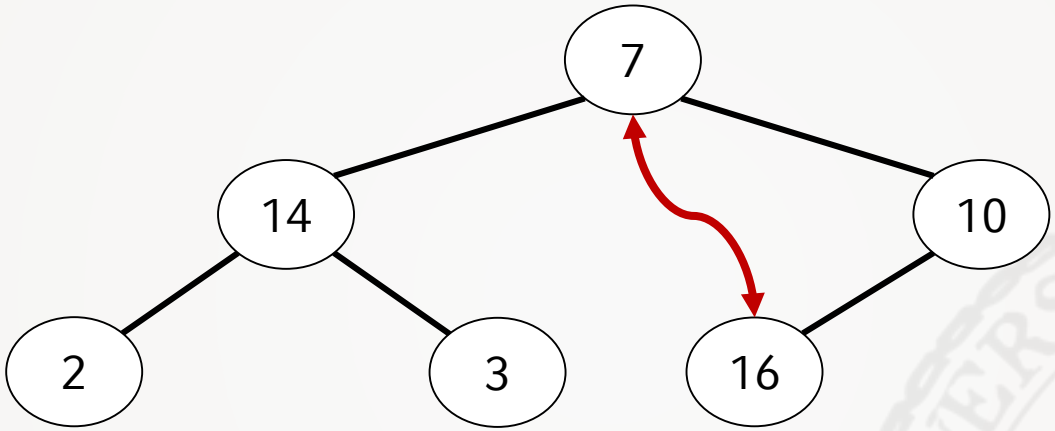
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



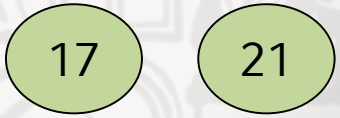
HeapSort: Beispiel

7	14	10	2	3	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



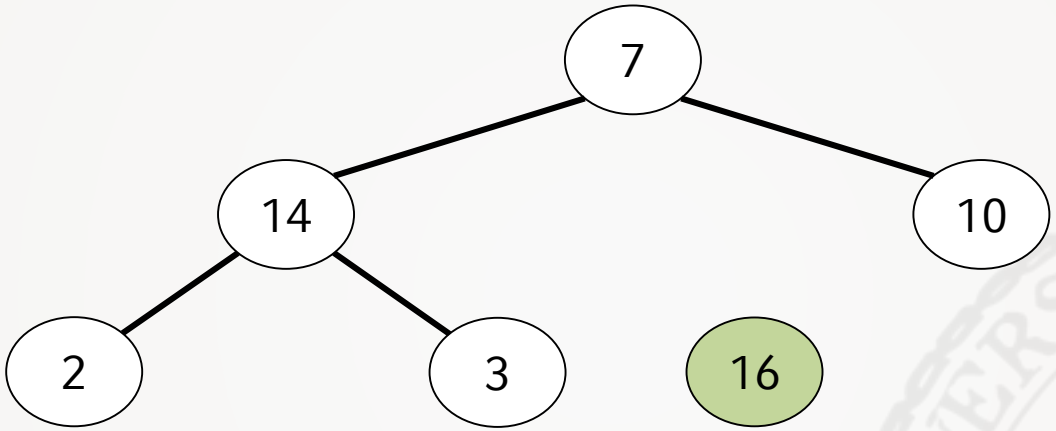
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



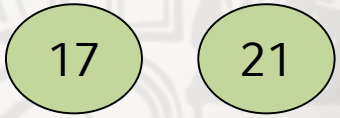
HeapSort: Beispiel

7	14	10	2	3	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



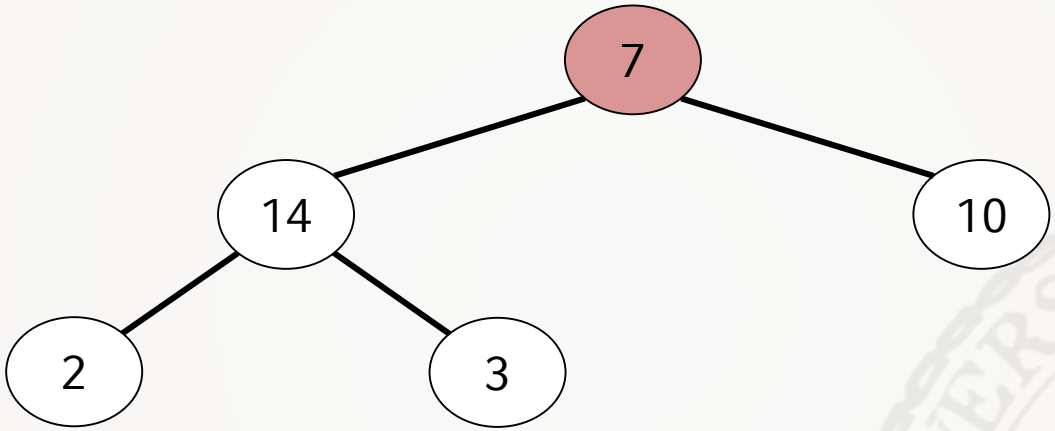
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



HeapSort: Beispiel

7	14	10	2	3	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

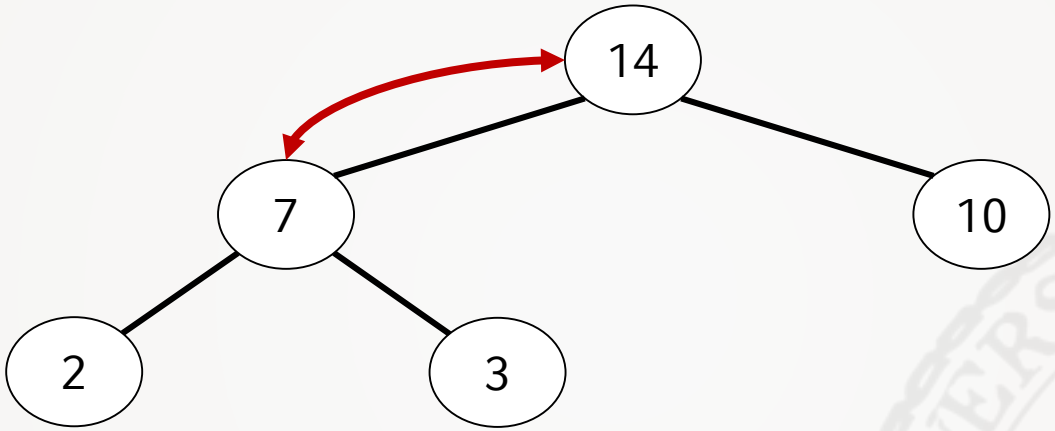


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

14	7	10	2	3	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

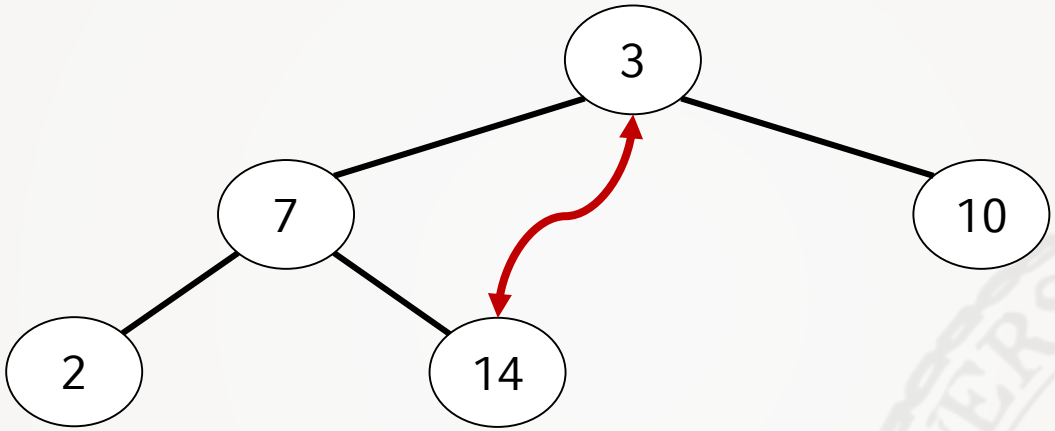


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

3	7	10	2	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element

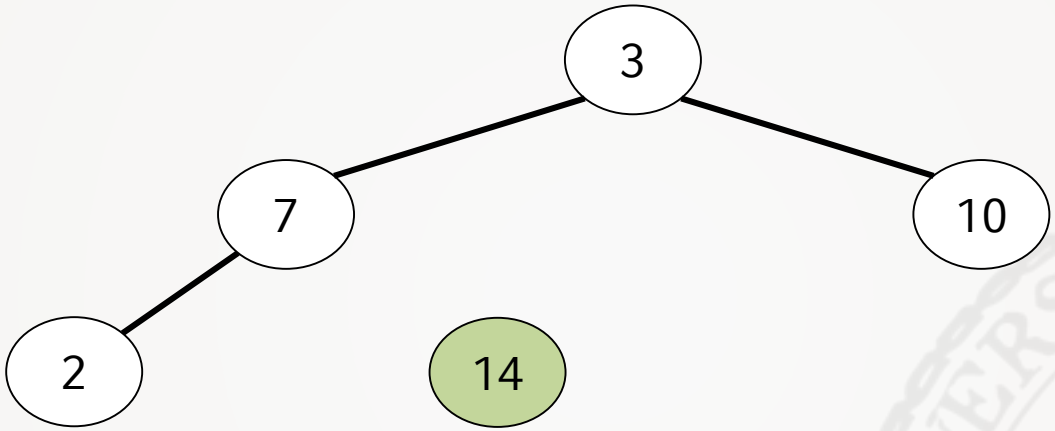


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

3	7	10	2	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur

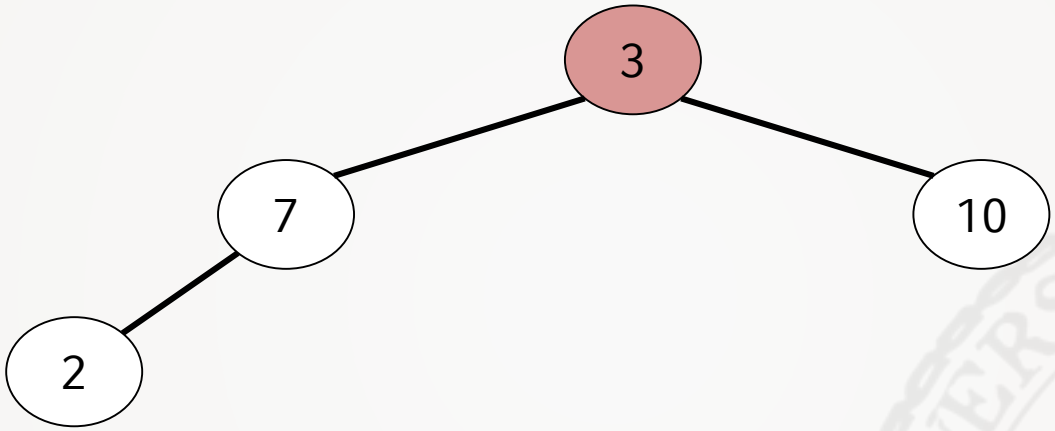


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

3	7	10	2	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

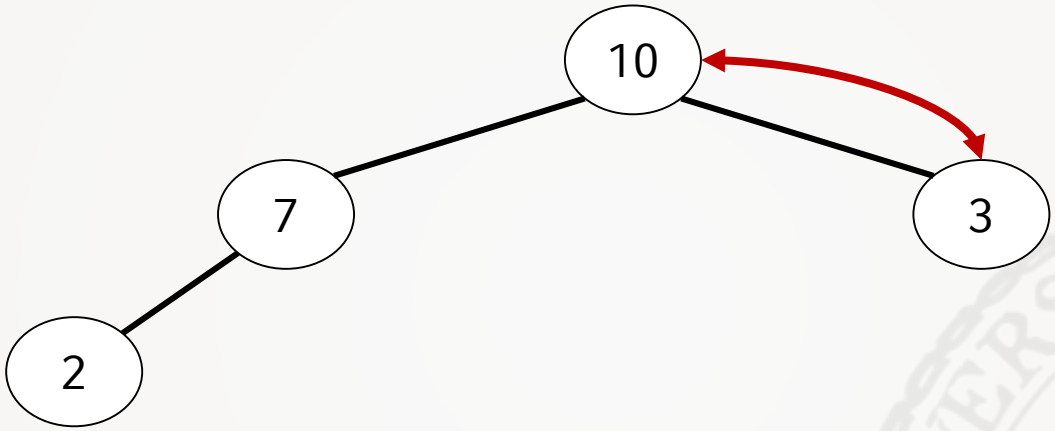


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

10	7	3	2	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

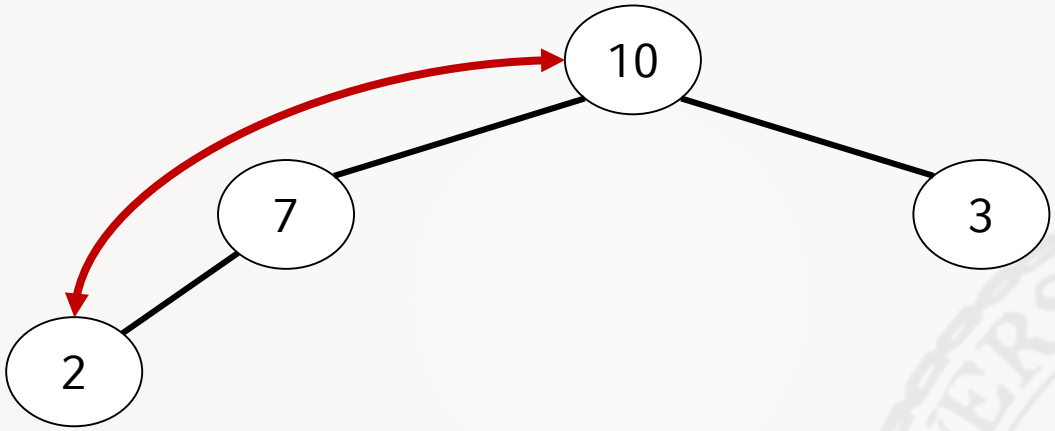


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

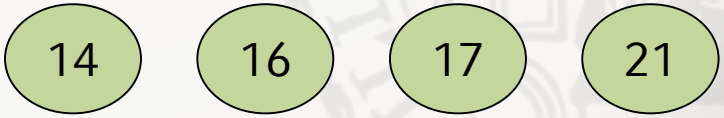
HeapSort: Beispiel

10	7	3	2	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



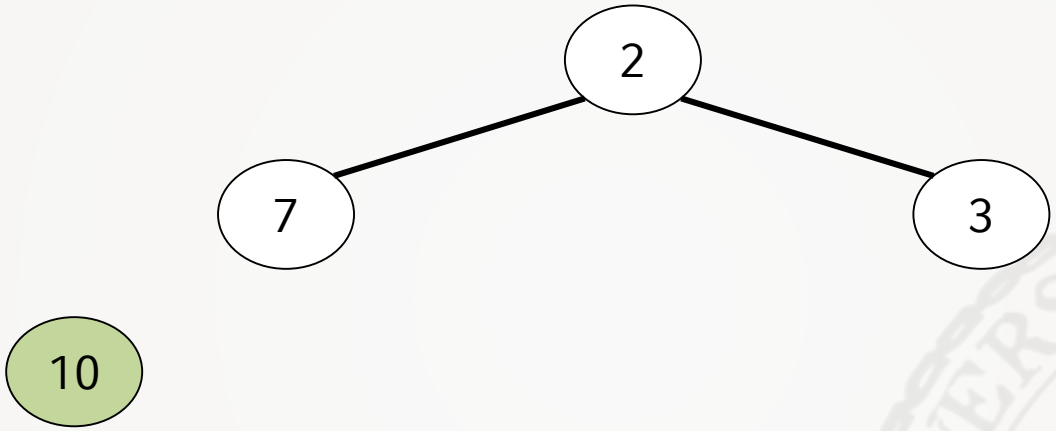
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



HeapSort: Beispiel

2	7	3	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur

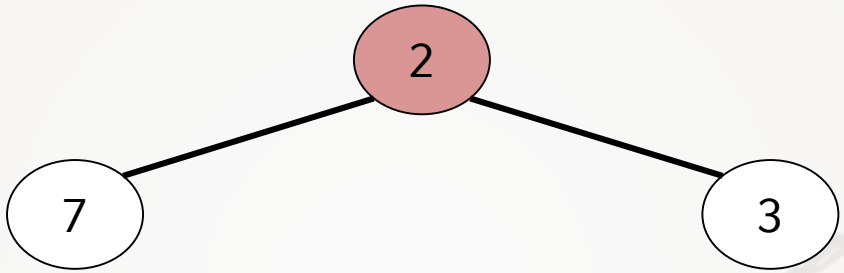


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

2	7	3	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

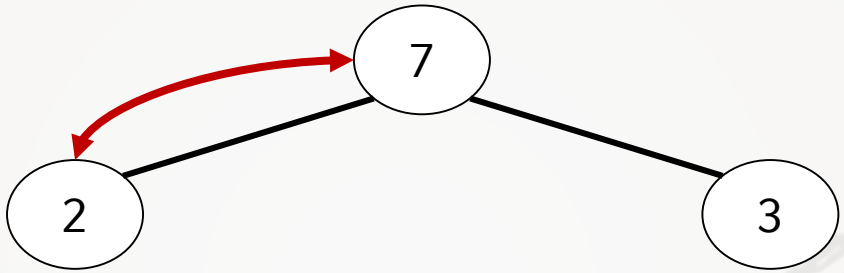


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

7	2	3	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen

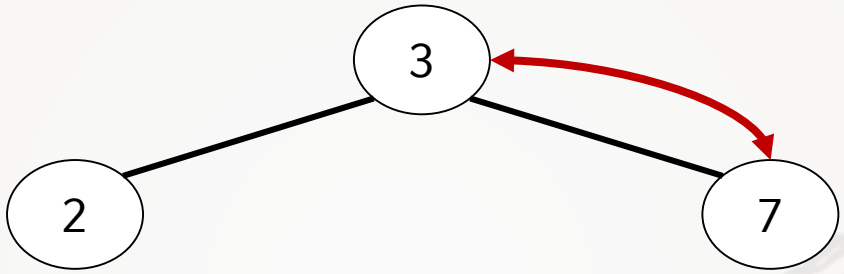


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

3	2	7	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element

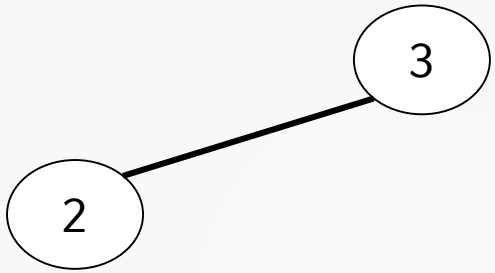


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

3	2	7	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur

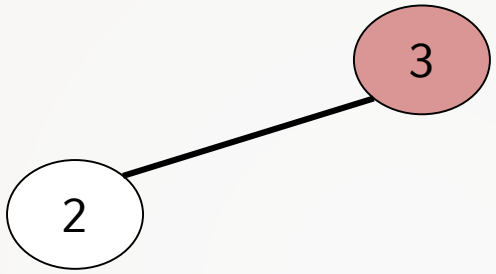


Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

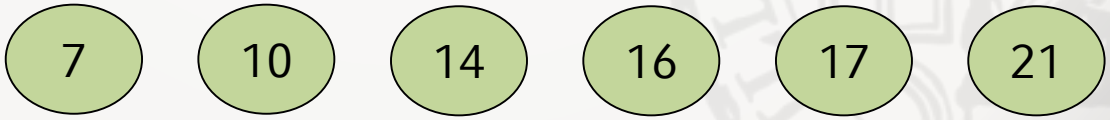
HeapSort: Beispiel

3	2	7	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



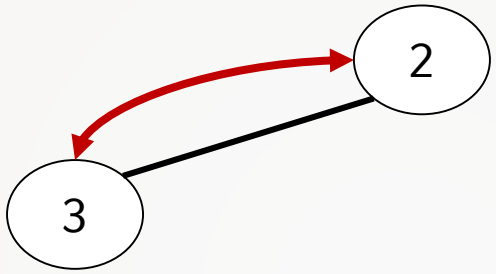
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



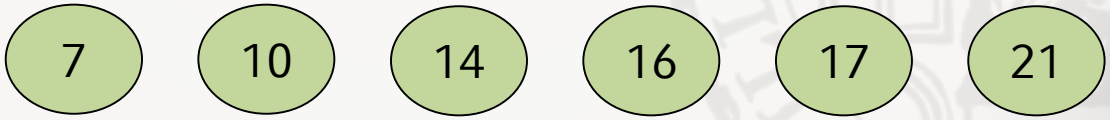
HeapSort: Beispiel

2	3	7	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



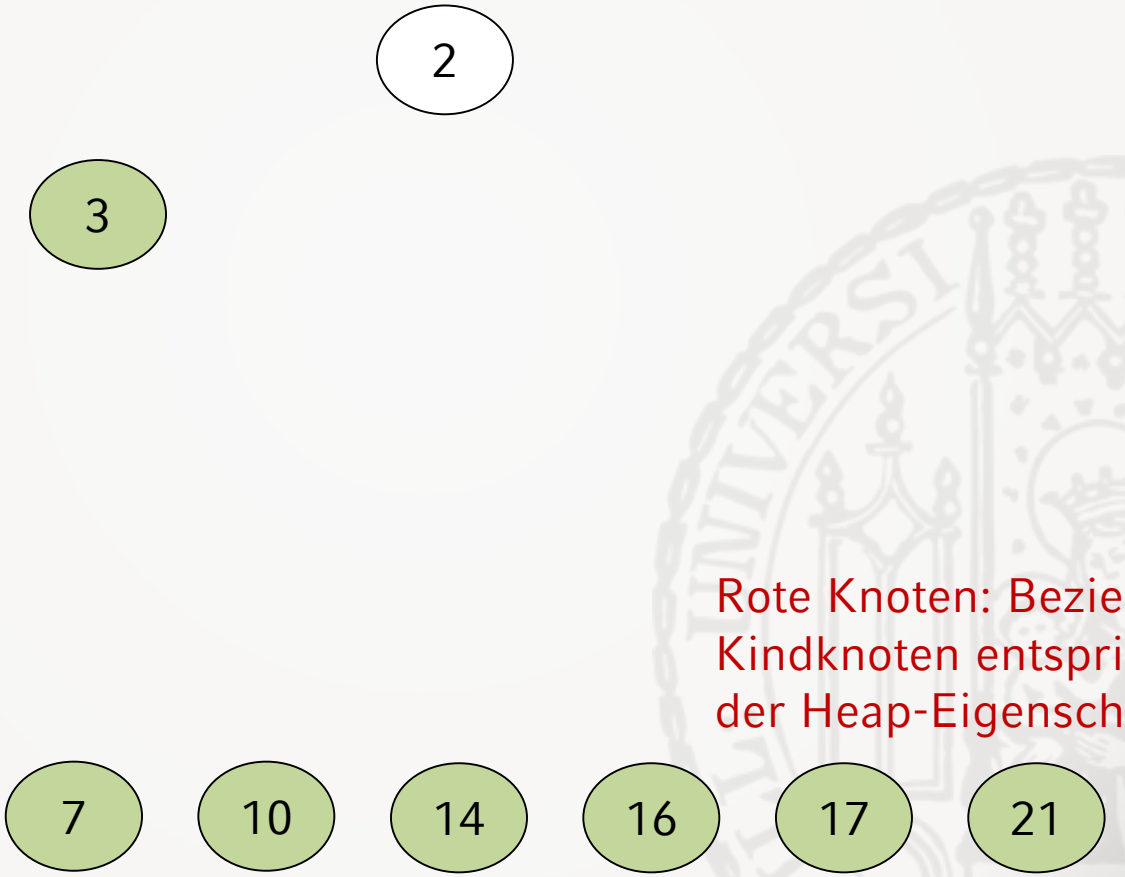
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



HeapSort: Beispiel

2	3	7	10	14	16	17	21

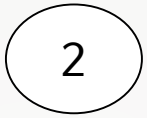
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne vorletzten Knoten aus der Struktur



HeapSort: Beispiel

2	3	7	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



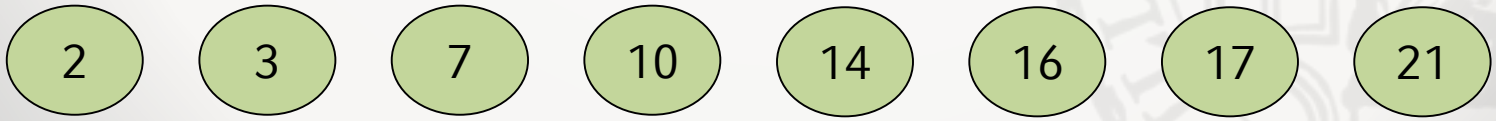
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

2	3	7	10	14	16	17	21

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Alle Knoten abgearbeitet und Sortieren beendet.

Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

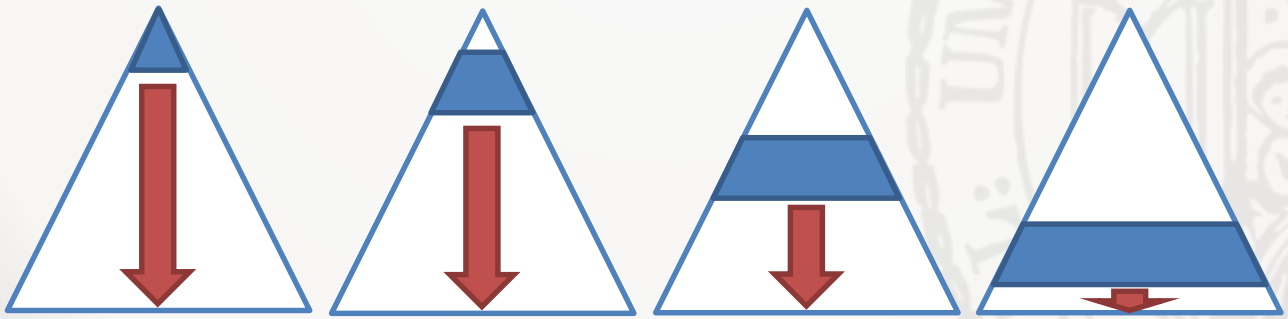


HeapSort: Beispiel mit Arrayeinbettung

Heapaufbau	10	14	7	17	3	21	2	16
	10	14	21	17	3	7	2	16
	10	17	21	16	3	7	2	14
	21	17	10	16	3	7	2	14
Sortierphase	14	17	10	16	3	7	2	21
	17	16	10	14	3	7	2	21
	2	16	10	14	3	7	17	21
	16	14	10	2	3	7	17	21
	7	14	10	2	3	16	17	21
	14	7	10	2	3	16	17	21
	3	7	10	2	14	16	17	21
	10	7	3	2	14	16	17	21
	2	7	3	10	14	16	17	21
	7	2	3	10	14	16	17	21
	3	2	7	10	14	16	17	21
	2	3	7	10	14	16	17	21

HeapSort: Heapaufbau

- Das erstmalige Aufstellen des Heap-Baums
- Naiv würde man raten:
  - Jeder Knoten sinkt maximal $\log_2 n$ tief ab.
  - Damit naiv: $O(n \log n)$
- Tatsächlich werden nur $O(n)$ viele Schritte benötigt. Intuitiv:
  - Nur die (wenigen) Knoten weit oben im Baum können überhaupt $\log_2 n$ viele Schritte nach unten versickern.
  - Die meisten Knoten befinden sich so tief im Baum, dass ein aufwändiges Versickern gar nicht möglich ist.



HeapSort: Komplexität Heapaufbau

- Betrachten wir einen vollständigen Binärbaum mit Höhe h und damit $n = 2^{h+1} - 1$ Knoten.
- Wir bestimmen den Aufwand des (möglichen) Versickern eines jeden Knotens bis zu den Blättern.
- Ein Blatt kann nicht versickern und erfüllt immer die Heapeigenschaft ⇒ 0 Vertauschungen für 2^h viele Knoten.
- Auf der nächsten Ebene gibt es 2^{h-1} Knoten, die maximal einen Schritt weit versickert werden.
- Nur die Wurzel kann h-mal versickern.
- Insgesamt, über alle Ebenen:

$$T(n) = \sum_{i=0}^{h} \left(i \cdot 2^{h-i} \right) = 2^h \cdot \sum_{i=0}^{h} \frac{i}{2^i}$$

HeapSort: Komplexität Heapaufbau (2)

- Kleiner Einschub: Geometrische Reihe ($|x| < 1$)

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

- Beide Seiten ableiten und multiplizieren mit x:

$$\sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

- Für $x = \frac{1}{2}$ erhalten wir:

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

HeapSort: Komplexität Heapaufbau (3)

Nun können wir unsere Formel

$$T(n) = \sum_{i=0}^{h} \left(i \cdot 2^{h-i} \right) = 2^h \cdot \sum_{i=0}^{h} \frac{i}{2^i}$$

abschätzen, indem wir noch ein paar (unendlich viele) Summanden hinzufügen:

$$2^h \cdot \sum_{i=0}^{h} \frac{i}{2^i} \;\leq\; 2^h \cdot \sum_{i=0}^{\infty} \frac{i}{2^i} \;\leq\; 2^h \cdot 2$$

Damit gilt mit $n = 2^{h+1} - 1$:

$$T(n) \;\leq\; 2^{h+1} \;=\; n + 1 \;\in\; O(n)$$

HeapSort: Komplexität

- Annahme: Daten liegen in Array vor.
- Herstellen der Heap-Eigenschaft („Heapify“): $O(n)$
- Für jeden der n Knoten (weil jeder mal Wurzel wird)
  - Tauschen mit dem letzten Knoten $O(1)$
  - Heapeigenschaft reparieren, Vertauschungen entlang eines Pfades im Baum: $O(\log n)$
- Insgesamt also $O(n \log n)$

Schranken des Sortierproblems

- Bisher: Sortieralgorithmen haben Komplexität
- Können wir einen Sortieralgorithmus finden, der besser ist als die gezeigten?
- Jetzt also: Frage nach der Komplexität des Problems

- Wie groß ist die Anzahl der Schlüsselvergleiche $T_{min}(n)$, um eine n-elementige Folge von Schlüsselelementen mit einem beliebigen (effizienten) Algorithmus zu sortieren?

- Existiert $T_{min}(n)$, sodass $T_{min}(n) \leq T_A(n)$ für alle Algorithmen A? D.h. jeder denkbare Algorithmus braucht in diesem Fall mindestens $T_{min}(n)$ viele Vergleiche.

Obere Schranke

- Betrachten wir beispielsweise MergeSort, so wissen wir bereits:

$$T_{min}(n) \leq n\lceil \log_2 n \rceil - n + 1 \in O(n \log n)$$

Untere asymptotische Schranke

- Wir benötigen zusätzlich ein Komplexitätsmaß für eine untere asymptotische Schranke.
- Erinnerung:
  $O(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$

- Analog zu $O(f)$ definieren wir:
  $\Omega(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0: |g(n)| \geq c \cdot |f(n)|\}$

  - „f wächst mindestens so schnell wie g“
  - „g ist untere Schranke für f“

Permutationen

- Gegeben sei eine n-elementige Folge

$$X = \langle x_1, x_2, \ldots, x_n \rangle$$

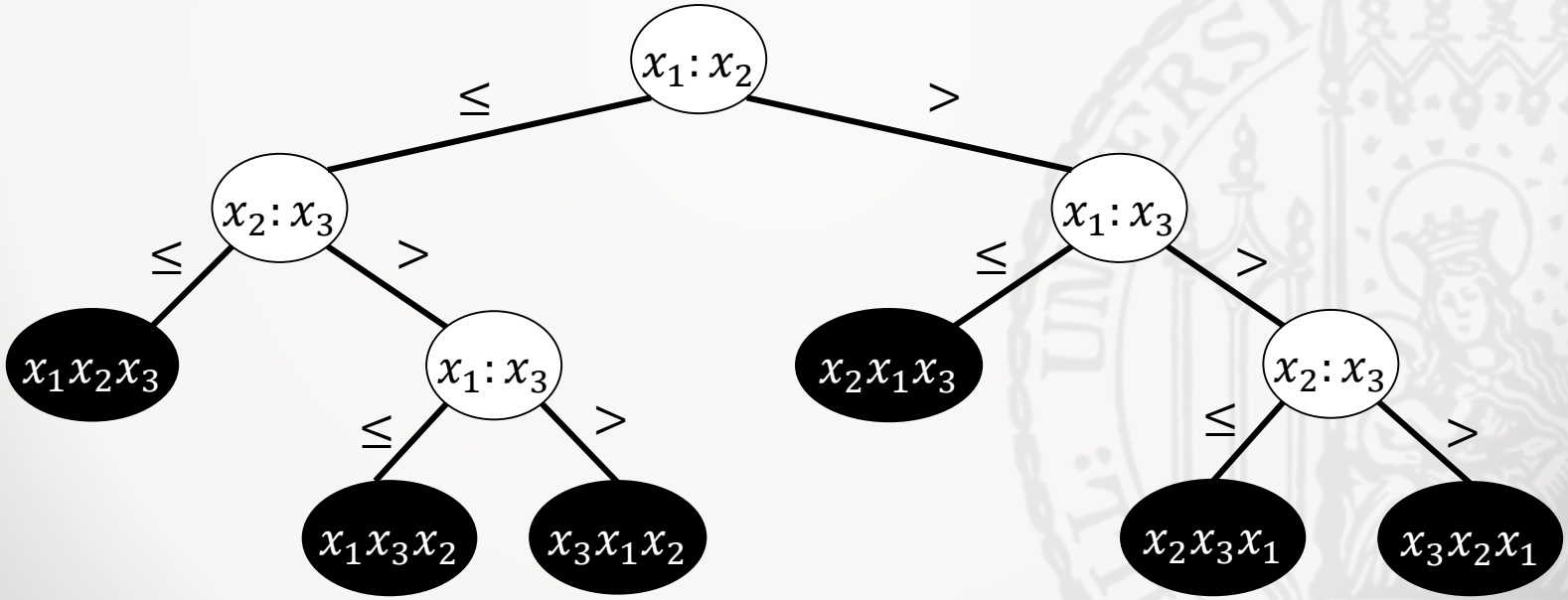
- Sei $\mathfrak{I}_n = \{1, 2, \ldots, n\} \subseteq \mathbb{N}$ eine Indexmenge. Eine Permutation π ist eine bijektive Abbildung $\pi: \mathfrak{I}_n \to \mathfrak{I}_n$.
- Im Beispiel bezeichne $\pi_i = \pi(i)$:
  3,2,1 ist Permutation von 1,2,3 mit $\pi_3 = 1, \pi_2 = 2,\ \pi_1 = 3$
- Es gibt $n!$ viele verschiedene Permutationen auf $\mathfrak{I}_n$.

- Sortieren ist die Auswahl einer Permutation X_π der Folge X mit

$$x_{\pi_1} \leq x_{\pi_2} \leq \cdots \leq x_{\pi_n}$$

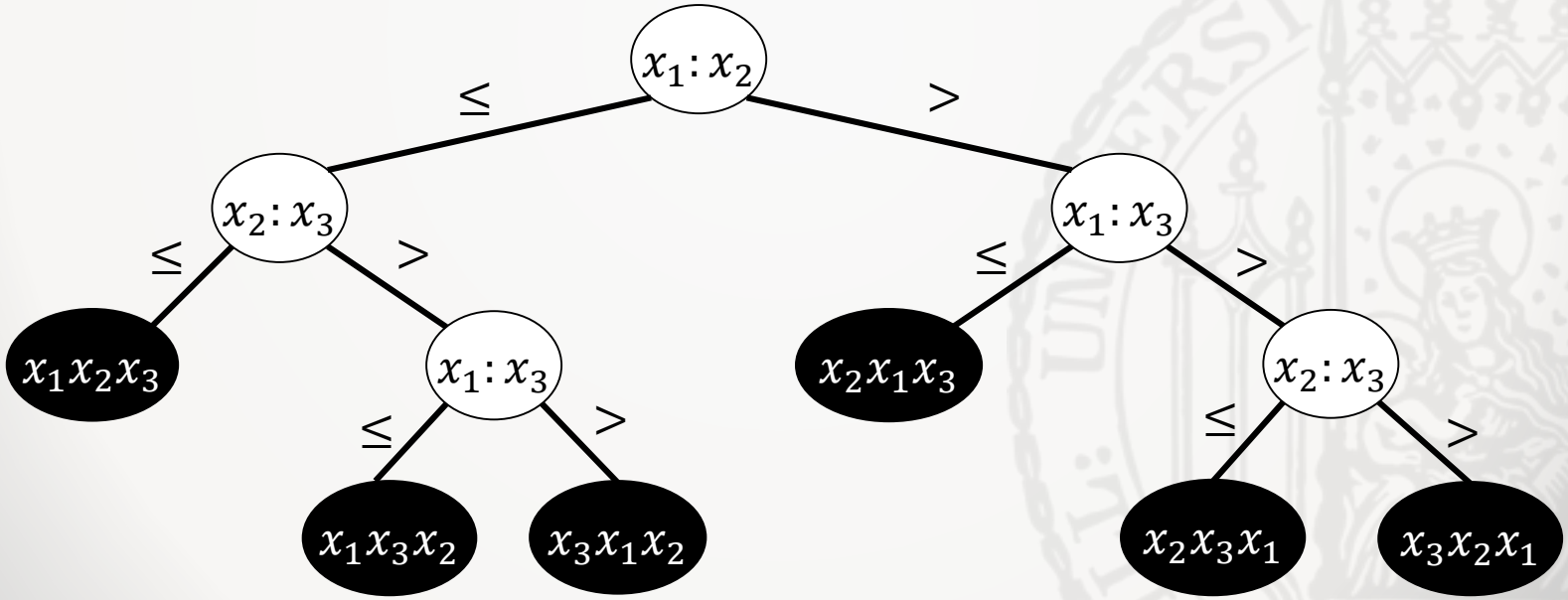
Entscheidungsbaum

- Der Ablauf eines nur auf Vergleichen basierenden Sortieralgorithmus kann durch einen Entscheidungsbaum dargestellt werden.
- Der folgende binäre Entscheidungsbaum „sortiert“ eine 3-elementige Folge $x_1x_2x_3$.
  Wegen 3! = 6 muss der Entscheidungsbaum 6 Blätter besitzen.



Entscheidungsreihenfolge von Sortieralgorithmen

- Das Entscheidungsbaummodell stellt unabhängig vom Algorithmus alle möglichen Ausgänge der Sortierung dar.
- Sortieralgorithmen müssen auf dieser Abstraktion das Ergebnis (= das Blatt mit der gesuchten Permutation) erreichen.
- Sie unterscheiden sich durch die Suchstrategie und damit dem Pfad von untersuchten Knoten in diesem Baum bis zu einem Blatt.



Untere Schranke für den Worst-Case

- Die Anzahl der Vergleiche im Worst-Case eines Algorithmus entspricht der Länge des längsten Weges im Entscheidungsbaum.
- Dies wiederum ist die Höhe des Entscheidungsbaums.
- Der Entscheidungsbaum habe Höhe h und b Blätter.
- Wir wissen:
  - Jede Permutation ist als Blatt erreichbar $\Rightarrow n! \leq b$
  - Ein Baum der Höhe h hat maximal 2^h Blätter $\Rightarrow b \leq 2^h$

$$2^h \geq b \geq n! \Rightarrow h \geq \log_2 n! \geq \log_2 \left(\frac{n}{e}\right)^n$$

$$= n \log_2 n - n \log_2 e$$

- Damit gilt $\exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall\, n > n_0$:

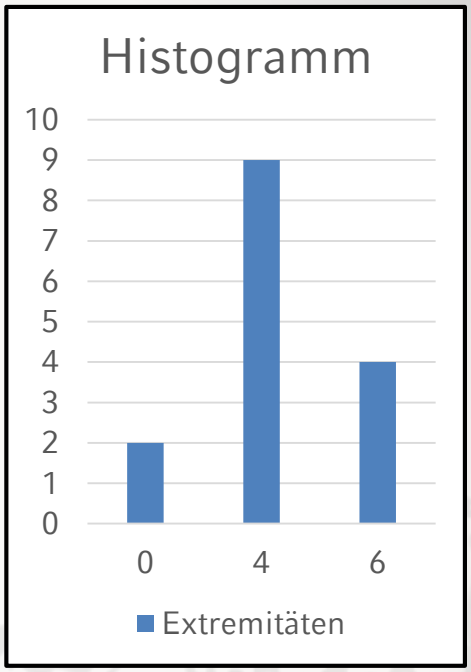
$$h > c(n \log n)$$

- Damit folgt der zentrale Satz:
  Die Anzahl der benötigten Vergleiche in jedem vergleichsbasierten Sortierverfahren wächst im Worst-Case mindestens wie $n \log n$.

CountingSort

- Angenommen, die Schlüssel sind darstellbar als ganzzahlige Werte im Bereich $0, \ldots, m-1$.
- Beispiel: Sortieren aller eingeschriebener Studenten anhand Studiensemesters $1 \ldots 20$.
- Wir können ein Histogramm erstellen und zählen für jeden Schlüsselwert seine Häufigkeit.
- Diese Häufigkeit liefert die Position für jeden Record.
- Bewege nun die Records an ihre errechnete Position.

CountingSort: Beispiel



- Beispiel: Sortiere Tiere anhand der Anzahl ihrer Extremitäten.
  Ameise, **B**är, **D**achs, **E**lefant, **F**liege, **G**iraffe, **H**ase, **K**arpfen, **L**öwe, **M**istkäfer, **S**chlange, **T**iger, **U**hu, **W**espe, **Z**ebra
- Mögliche Kategorien: 0, 4, 6

- Damit liegen die Bereiche im Zielarray fest:

0	4	6

- Ein weiterer Durchlauf reicht, um die einzelnen Schlüssel an die passende Position zu schreiben:

K	S	B	D	E	G	H	L	T	U	Z	A	F	M	W

CountingSort: Komplexität

- Es werden n Schlüssel sortiert mit m verschiedenen Schlüsselwerten
- Initialisierung des Histogramms → $O(m)$
- Zählen der Schlüssel, Schleife über alle Schlüssel → $O(n)$
- Bestimmung der Adressen für alle m Werte → $O(m)$
- Zusammenstellung des finalen sortierten Arrays $O(n)$

- Insgesamt: $O(m+n) = O(\max(m,n))$

- Für kleine m ist CountingSort besser als vorherige Verfahren?
  - Untere Schranke $O(n \log n)$ beruht auf der Anzahl der Vergleiche.
  - CountingSort ist nicht vergleichsbasiert, sondern adressbasiert.

Modernes effizientes Sortieren

- Java: Arrays.sort() nutzt „Dual-Pivot Quicksort“
  - Wähle zwei Pivots und splitte in drei Teile
- Vergleichsbasierte Methoden oft als Hybridverfahren
  - IntroSort sortiert wie QuickSort, außer eine Bewertung im Teilschritt prognostiziert den Worst-Case, dann wird HeapSort genutzt
- Python: Timsort
  - Hybrid aus MergeSort und InsertionSort
  - Findet vorsortierte Teilstücke und sortiert bloß dazwischen
- Oft anwendungsspezifisch:
  - Bei großer Wahrscheinlichkeit von Vorsortierung ist Best-Case wichtiger als Worst-Case
- Paralleles Sortieren ebenfalls möglich. Keine Verbesserung der Komplexität, aber Beschleunigung durch zeitgleiches Rechnen.

Fazit: Sortieren

- Einfache Sortieralgorithmen (BubbleSort, SelectionSort, InsertionSort):
  - Einfache Implementierung
  - Im schlimmsten Fall unnötig viele Vergleiche $\Rightarrow O(n^2)$
- Bessere Suchverfahren (QuickSort, MergeSort, HeapSort):
  - Komplexere Implementierung
  - Nutzen Datenstrukturen zur Verwaltung von Wissen über vorherige Vergleiche
  - Daher kann Sortieren in $O(n \log n)$ durchgeführt werden.
- Verfahren, die auf elementweisen Vergleichen beruhen, brauchen $\Omega(n \log n)$ viele Vergleiche.