

# Kapitel ADS:III

## III. Sortieren und Suchen

- Sortieralgorithmen
- Insertion Sort
- Merge Sort
- Heapsort
- Quicksort
- Minimales vergleichsbasiertes Sortieren
- Counting Sort
- Radix Sort
- Bucket Sort
- Suchen

# Sortieralgorithmen

## Sortierproblem

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Hintergrund:

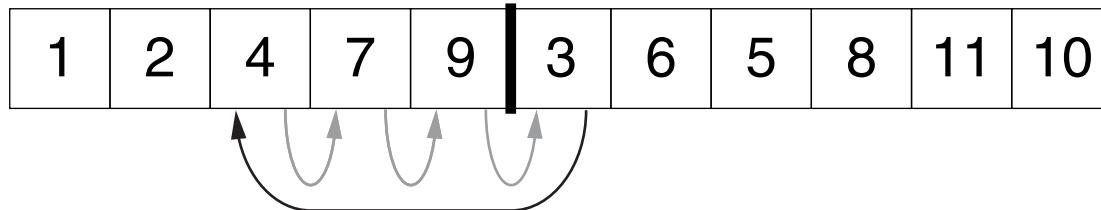
- Die Problemdefinition ist reduziert auf das Wesentliche: **Zahlen** sortieren.
- In der Praxis wollen wir aber kompliziertere Datentypen verarbeiten.
- Der erste Schritt dafür ist, eine interessante Eigenschaft der Daten zu betrachten: z.B. Größe, Wert, Wichtigkeit, Erstelldatum, etc.
- Diese Eigenschaft wird quantifiziert und die Daten danach sortiert.
- Die interessierende Eigenschaft wird als **Sortierschlüssel** bezeichnet.
- Zu analysierende Daten werden mit Sortierschlüsseln verknüpft gespeichert.
- Der Einfachheit halber blenden wir die verknüpften Daten aus.

# Sortieralgorithmen

Sortierparadigmen [Knuth 2003]

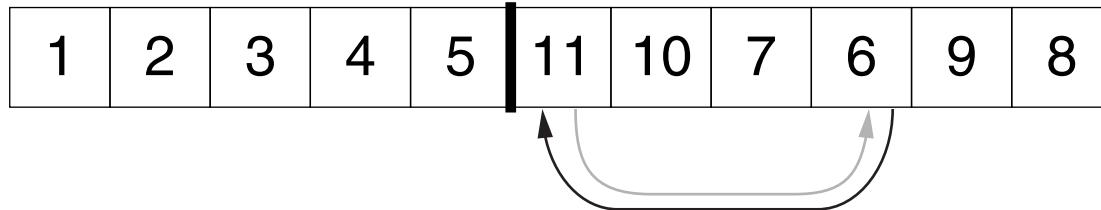
## 1. Einfügen (*Insertion*)

Das  $i$ -te Element wird in die  $i - 1$  zuvor sortierten Elemente eingefügt.



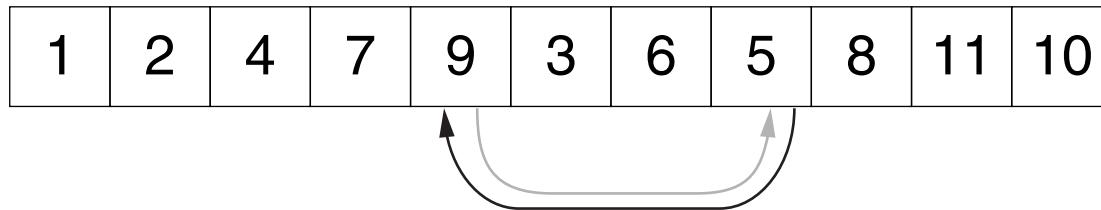
## 2. Selektieren (*Selection*)

Das nächstkleinste (-größte) Element wird den zuvor sortierten Elementen angefügt.



## 3. Vertauschen (*Exchanging*)

Pärchen von Elementen werden vertauscht, wenn sie falsch sortiert sind.

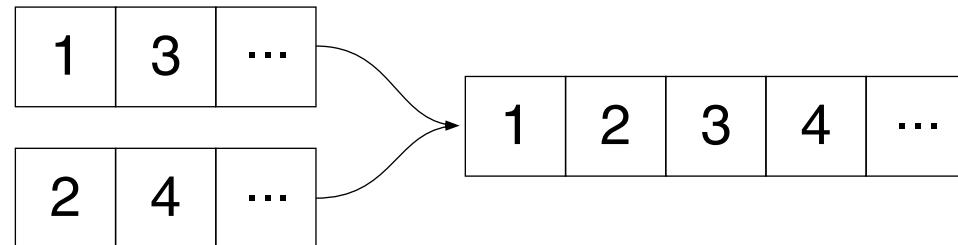


# Sortieralgorithmen

Sortierparadigmen [Knuth 2003] (Fortsetzung)

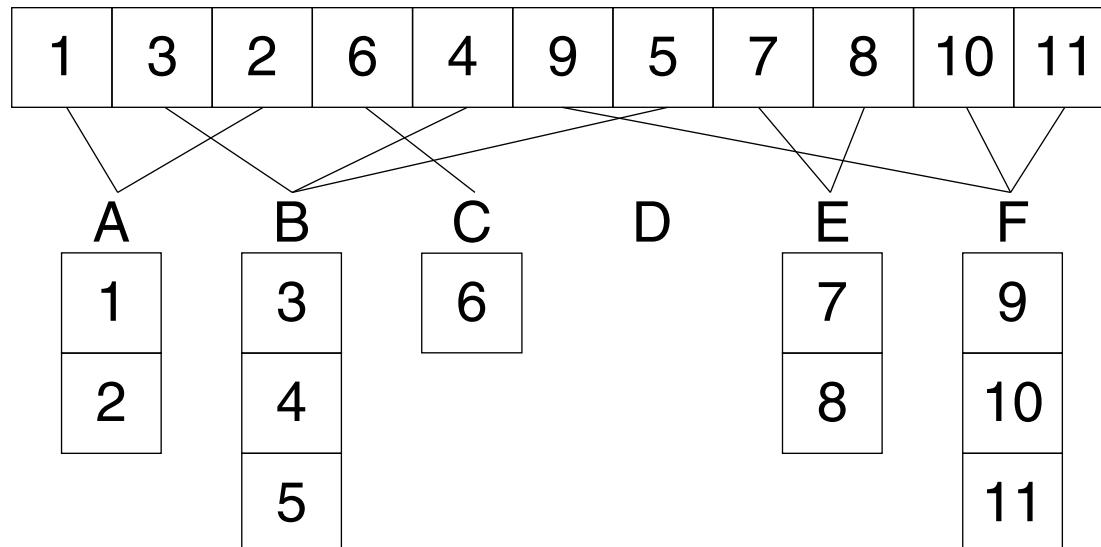
## 4. Vereinigen (*Merging*)

Kombination mehrerer sortierter Folgen von Elementen zu einer sortierten Folge.



## 5. Verteilen (*Distribution*)

Jedes Element wird auf vordefinierte Mengen verteilt und die Mengen dann gesammelt.



# Sortieralgorithmen

## Überblick

Algorithmus	Laufzeitkomplexität		Platz	In-place	Stabil	Paradigma
	Average	Worst				
Insertion sort	$n^2$	$n^2$	1	ja	ja	Einfügen
Shell sort	n/a	$n \lg^2 n$	1	ja	nein	Einfügen
Merge sort	$n \lg n$	$n \lg n$	$n$	nein	ja	Vereinigen
Selection sort	$n^2$	$n^2$	1	ja	nein	Selektion
Tournament sort	$n \lg n$	$n \lg n$	$n$	nein	nein	Selektion
Heapsort	$n \lg n$	$n \lg n$	1	ja	nein	Selektion
Bubble sort	$n^2$	$n^2$	1	ja	ja	Vertauschen
Quicksort	$n \lg n$	$n^2$	$n$	nein	nein	Vertauschen
Counting Sort	$n + k$	$n + k$	$n + k$	nein	ja	Verteilen
Radix Sort	$d(n + k)$	$d(n + k)$	$n$	nein	ja	Verteilen
Bucket Sort	$n$	$n^2$	$n + k$	nein	ja	Verteilen

$n$  = Anzahl Elemente;  $k$  = Anzahl möglicher Werte;  $d$  = Zahl der Stellen des längsten Elements.

# Insertion Sort

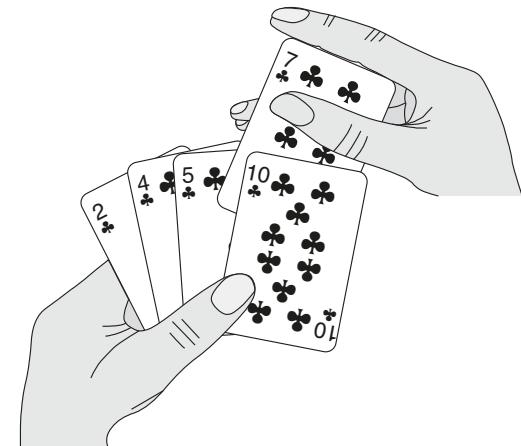
## Algorithmus

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortiere analog zum Aufnehmen und Einsortieren von Handkarten beim Karten spielen.

Anweisungen zum Karten sortieren:

- Eine Karte vom Stapel aufnehmen.
- Kartenwert merken.
- Angefangen bei der Karte rechts auf der Hand:
  - Prüfe, ob sie einen höheren Wert hat.
  - Wenn ja, schiebe sie nach rechts auf der Hand.
  - Dann gehe zur links benachbarten Karte.
  - Fahre fort, solange höherwertige Karten da sind.
- Füge die Karte an der nun freien Stelle ein.
- Fahre fort, solange Karten auf dem Stapel sind.  
→ Die Handkarten sind sortiert.



# Insertion Sort

## Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von  $n$  Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Datenstruktur Array:

- $n$  gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes  $A[i]$

Beispiel:

	1	2	3	4	5	6
A	5	2	4	6	1	3

# Insertion Sort

## Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von  $n$  Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

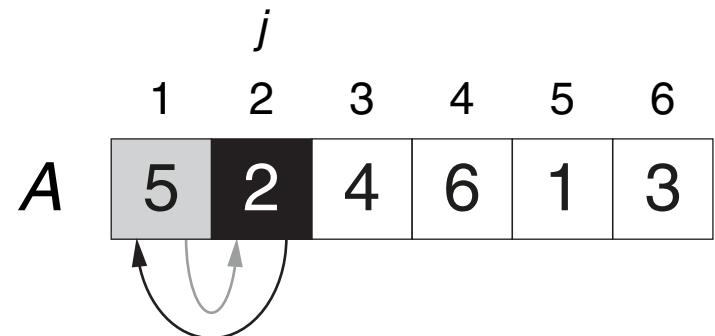
*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Datenstruktur Array:

- $n$  gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes  $A[i]$

Beispiel:



# Insertion Sort

## Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von  $n$  Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

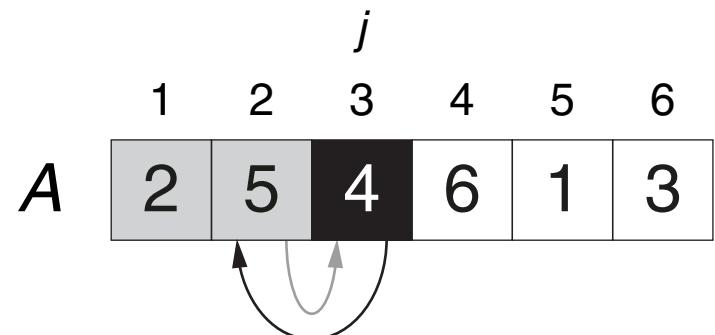
*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Datenstruktur Array:

- $n$  gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes  $A[i]$

Beispiel:



# Insertion Sort

## Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von  $n$  Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

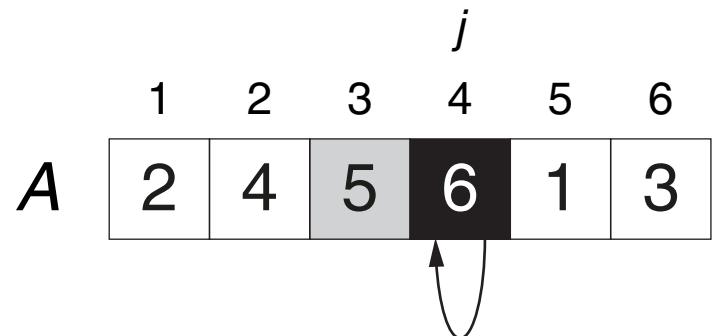
*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Datenstruktur Array:

- $n$  gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes  $A[i]$

Beispiel:



# Insertion Sort

## Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von  $n$  Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

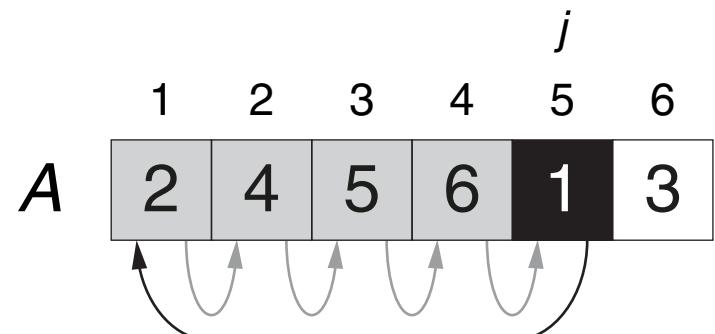
*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Datenstruktur Array:

- $n$  gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes  $A[i]$

Beispiel:



# Insertion Sort

## Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von  $n$  Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

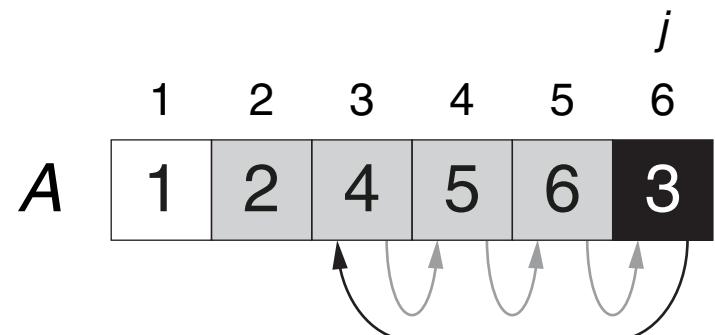
*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Datenstruktur Array:

- $n$  gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes  $A[i]$

Beispiel:



# Insertion Sort

## Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von  $n$  Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Datenstruktur Array:

- $n$  gleich große Speicherplätze
- Alle Werte vom gleichen Typ
- Wahlfreier Zugriff über Indizes  $A[i]$

Beispiel:

	1	2	3	4	5	6
A	1	2	3	4	5	6

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$T(n) = c_1 \cdot n$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$T(n) = c_1 \cdot n + c_2 \cdot (n - 1)$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \sum_{j=2}^n t_j \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \sum_{j=2}^n t_j \\& + c_5 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_6 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_7 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \sum_{j=2}^n t_j \\& + c_5 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_6 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_7 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Best Case:

$$t_j = 1 \quad \leadsto \quad \sum_{j=2}^n 1 = n - 1$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot (n - 1) \\& + c_5 \cdot 0 \\& + c_6 \cdot 0 \\& + c_7 \cdot 0 \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Best Case:

$$t_j = 1 \quad \leadsto \quad \sum_{j=2}^n 1 = n - 1$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$T(n) = c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot (n - 1) + c_4 \cdot (n - 1) + c_5 \cdot 0 + c_6 \cdot 0 + c_7 \cdot 0 + c_8 \cdot (n - 1) + c_9 \cdot (n - 1)$$

Best Case:

$$t_j = 1 \quad \leadsto \quad \sum_{j=2}^n 1 = n - 1$$

	1	2	3	4	5	6
$A$	1	2	3	4	5	6

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot (n - 1) \\& + c_5 \cdot 0 \\& + c_6 \cdot 0 \\& + c_7 \cdot 0 \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Best Case:

$$T(n) = \underbrace{(c_1 + c_2 + c_3 + c_4 + c_8 + c_9)}_a \cdot n - \underbrace{(c_2 + c_3 + c_4 + c_8 + c_9)}_b$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot (n - 1) \\& + c_5 \cdot 0 \\& + c_6 \cdot 0 \\& + c_7 \cdot 0 \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Best Case:

$$T(n) = a \cdot n + b \quad (\text{lineare Funktion})$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \sum_{j=2}^n t_j \\& + c_5 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_6 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_7 \cdot \sum_{j=2}^n (t_j - 1) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Worst Case:

$$t_j = j \quad \leadsto \quad \sum_{j=2}^n j = \frac{n \cdot (n + 1)}{2} - 1$$

	1	2	3	4	5	6
$A$	6	5	4	3	2	1

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \left(\frac{n \cdot (n+1)}{2} - 1\right) \\& + c_5 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_6 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_7 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Worst Case:

$$t_j = j \quad \leadsto \quad \sum_{j=2}^n j = \frac{n \cdot (n + 1)}{2} - 1$$

	1	2	3	4	5	6
$A$	6	5	4	3	2	1

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \left(\frac{n \cdot (n+1)}{2} - 1\right) \\& + c_5 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_6 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_7 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Worst Case:

$$T(n) = \underbrace{\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)}_a \cdot n^2 + \underbrace{\left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 + c_9\right)}_b \cdot n - \underbrace{\left(c_2 + c_3 + c_4 + c_8 + c_9\right)}_c$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \left(\frac{n \cdot (n+1)}{2} - 1\right) \\& + c_5 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_6 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_7 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

Worst Case:

$$T(n) = a \cdot n^2 + b \cdot n + c \quad (\text{quadratische Funktion})$$

# Insertion Sort

## Laufzeitanalyse

*InsertionSort(A)*

1. **FOR**  $j = 2$  **TO**  $n$  **DO**
2.      $a_j = A[j]$
3.      $i = j - 1$
4.     **WHILE**  $i > 0$  **AND**  $A[i] > a_j$  **DO**
5.          $A[i + 1] = A[i]$
6.          $i = i - 1$
7.     **ENDDO**
8.      $A[i + 1] = a_j$
9. **ENDDO**

Laufzeitfunktion:

$$\begin{aligned}T(n) = & c_1 \cdot n \\& + c_2 \cdot (n - 1) \\& + c_3 \cdot (n - 1) \\& + c_4 \cdot \left(\frac{n \cdot (n+1)}{2} - 1\right) \\& + c_5 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_6 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_7 \cdot \left(\frac{(n-1) \cdot n}{2}\right) \\& + c_8 \cdot (n - 1) \\& + c_9 \cdot (n - 1)\end{aligned}$$

- Die Laufzeit von Insertion Sort ist nach unten beschränkt durch eine lineare Funktion und nach oben durch eine quadratische Funktion.
- Die Laufzeit von Insertion Sort wächst **asymptotisch** höchstens wie  $g(n) = n^2$ .

## Bemerkungen:

- Wir nehmen an, dass jede Zeile  $i$  des Algorithmus eine konstante Zeit  $c_i$  benötigt.
- Wenn eine For- oder While-Schleife auf normale Weise wegen Nichterfüllung der Schleifenbedingung endet, wurde die Schleifenbedingung ein Mal häufiger ausgeführt als der Schleifenrumpf.
- Für  $j \in [2, n]$  entspricht  $t_j$  der Anzahl der Iterationen der While-Schleife.
- Nur Zeilen 4-7 sind abhängig von der Struktur der Probleminstanz. Die übrigen Zeilen hängen nur von der Größe der Probleminstanz  $n$  ab.
- Minimal wird  $t_j$  genau dann, wenn die While-Schleife für  $j$  nicht durchlaufen werden muss. Das geschieht, wenn eine der Schleifenbedingungen nicht erfüllt ist, was nur dann der Fall ist, wenn das aktuell einzusortierende Element  $A[j]$  bereits an der richtigen Stelle steht.
- Maximal wird  $t_j$  genau dann, wenn die While-Schleife für  $j$  das Element  $A[j]$  mit allen  $j - 1$  vorangehenden Elementen vertauschen muss.
- Die arithmetische Reihe:  $\sum_{j=1}^n j = \frac{n \cdot (n+1)}{2}$ . Für  $k = j - 1$  ist  $\sum_{j=2}^n (j - 1) = \sum_{k=1}^{n-1} k = \frac{(n-1) \cdot n}{2}$ .
- Der Average Case entspricht dem Worst Case. Für eine Zufallsfolge von  $n$  Zahlen muss für  $j$  im Durchschnitt das halbe vorangehende Array durchsucht werden, um zu entscheiden, wo  $A[j]$  eingesortiert werden muss: Der Erwartungswert  $E(t_j) = j/2$ . Die Laufzeit ist also im Schnitt die Hälfte der Laufzeit des Worst Cases, also weiterhin eine quadratische Funktion.

# Merge Sort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der für jede Instanz  $A$  eine Lösung  $A'$  berechnet.

Idee: Rekursives Divide and Conquer

# Merge Sort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der für jede Instanz  $A$  eine Lösung  $A'$  berechnet.

Idee: Rekursives Divide and Conquer

Rekursives Sortieren eines Arrays  $A$  der Länge  $n$ :

- $n = 1$ :  $A$  ist sortiert.
- $n > 1$ : Sortiere  $A[1.. \lfloor n/2 \rfloor]$  und  $A[\lfloor n/2 \rfloor + 1..n]$  und vereinige sie anschließend.

Voraussetzung:

- Die Vereinigung zweier sortierter Arrays zu einem sortierten Array ist effizient.

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*(A, p, q)
4.     *MergeSort*(A, q + 1, r)
5.     *Merge*(A, p, q, r)
6. **ENDIF**

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*(A, p, q)
4.     *MergeSort*(A, q + 1, r)
5.     *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)	1	2	3	4	5	6	7	8	9	10	11
	4	7	2	6	1	4	7	3	5	2	6

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*(A, p, q)
4.     *MergeSort*(A, q + 1, r)
5.     *Merge*(A, p, q, r)
6. **ENDIF**

	1	2	3	4	5	6	7	8	9	10	11
(A, 1, 11)	4	7	2	6	1	4	7	3	5	2	6
(A, 1, 6)	4	7	2	6	1	4					

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*(A, p, q)
4.     *MergeSort*(A, q + 1, r)
5.     *Merge*(A, p, q, r)
6. **ENDIF**

	1	2	3	4	5	6	7	8	9	10	11
(A, 1, 11)	4	7	2	6	1	4	7	3	5	2	6
(A, 1, 6)	4	7	2	6	1	4					
(A, 1, 3)	4	7	2								

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

1	2	3	4	5	6	7	8	9	10	11
4	7	2	6	1	4	7	3	5	2	6
(A, 1, 11)										
4	7	2	6	1	4					
(A, 1, 6)										
4	7	2								
(A, 1, 3)										
4	7									
(A, 1, 2)										
4	7									

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)	1	2	3	4	5	6	7	8	9	10	11
	4	7	2	6	1	4	7	3	5	2	6
(A, 1, 6)	4	7	2	6	1	4					
(A, 1, 3)	4	7	2								
(A, 1, 2)	4	7									
(A, 1, 1)	4										

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*(A, p, q)
4.     *MergeSort*(A, q + 1, r)
5.     *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)	1	2	3	4	5	6	7	8	9	10	11
	4	7	2	6	1	4	7	3	5	2	6
(A, 1, 6)	4	7	2	6	1	4					
(A, 1, 3)	4	7	2								
(A, 1, 2)	4	7									
(A, 2, 2)	4	7									

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

	1	2	3	4	5	6	7	8	9	10	11
(A, 1, 11)	4	7	2	6	1	4	7	3	5	2	6
(A, 1, 6)	4	7	2	6	1	4					
(A, 1, 3)	4	7	2								
(A, 1, 1, 2)	4	7									
	4	7									

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)	1	2	3	4	5	6	7	8	9	10	11
	4	7	2	6	1	4	7	3	5	2	6
(A, 1, 6)	4	7	2	6	1	4					
(A, 1, 3)	4	7	2								
(A, 3, 3)	4	7	2								
	4	7									

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)	1	2	3	4	5	6	7	8	9	10	11
	4	7	2	6	1	4	7	3	5	2	6
(A, 1, 6)	4	7	2	6	1	4					
(A, 1, 2, 3)	2	4	7								
	4	7	2								
	4	7									

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)

(A, 1, 6)

(A, 4, 6)

1	2	3	4	5	6	7	8	9	10	11
4	7	2	6	1	4	7	3	5	2	6
4	7	2	6	1	4					
2	4	7	1	4	6					
4	7	2	1	6	4					
4	7		6	1						

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)

(A, 1, 3, 6)

1	2	3	4	5	6	7	8	9	10	11
4	7	2	6	1	4	7	3	5	2	6
1	2	4	4	6	7					
2	4	7	1	4	6					
4	7	2	1	6	4					
4	7		6	1						

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 11)

(A, 7, 11)

1	2	3	4	5	6	7	8	9	10	11
4	7	2	6	1	4	7	3	5	2	6
1	2	4	4	6	7	2	3	5	6	7
2	4	7	1	4	6	3	5	7	2	6
4	7	2	1	6	4	3	7	5	2	6
4	7		6	1		7	3			

# Merge Sort

## Algorithmus

Algorithmus: Merge Sort.

Eingabe: A. Array von  $n$  Zahlen.

p. Index ab dem A betrachtet wird.

r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*MergeSort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p + r)/2 \rfloor$
3.   *MergeSort*(A, p, q)
4.   *MergeSort*(A, q + 1, r)
5.   *Merge*(A, p, q, r)
6. **ENDIF**

(A, 1, 6, 11)

1	2	3	4	5	6	7	8	9	10	11
1	2	2	3	4	4	5	6	6	7	7
1	2	4	4	6	7	2	3	5	6	7
2	4	7	1	4	6	3	5	7	2	6
4	7	2	1	6	4	3	7	5	2	6
4	7		6	1		7	3			

## Bemerkungen:

- Die Teilung in Teilarrays erfolgt nicht durch Anlegen neuer Arrays, sondern durch Einschränkung der Parameter  $p$  und  $r$ , die das Intervall von  $A$  beschreiben, in dem der Algorithmus arbeiten soll. Das Array selbst wird als Referenzparameter übergeben.
- Die Funktion  $\text{Merge}(A, p, q, r)$  setzt voraus, dass die Teilarrays  $A[p..q]$  und  $A[(q + 1)..r]$  sortiert sind und vereinigt die beiden Teilarrays dann zu einem insgesamt sortierten Array.

# Merge Sort

## Merge

Problem: Sortierte Arrays Vereinigen

Instanz:  $L$  und  $R$ . Folgen von  $n_1$  und  $n_2$  aufsteigend sortierten Zahlen.

Lösung: Array  $A$  der Länge  $n_1 + n_2$ , dass die Zahlen aus  $L$  und  $R$  aufsteigend sortiert enthält.

Wunsch: Ein Algorithmus, der zwei Arrays vereinigt.

Idee: Verfahren zur Vereinigung zweier sortierter Kartenstapel umsetzen.

Algorithmus: Merge.

Eingabe: A. Array von  $n$  Zahlen.

$p, q, r$ . Indexe, so dass  $0 < p \leq q < r \leq n$  sowie  $A[p..q]$  und  $A[q + 1..r]$  sortierte Teilarrays bilden.

Ausgabe: Aufsteigend sortiertes Teilarray  $A[p..r]$ .

# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

1.  $n_1 = q - p + 1$
2.  $n_2 = r - p$
3.  $L = \text{array}(n_1 + 1)$
4.  $R = \text{array}(n_2 + 1)$
5. **FOR**  $i = 1$  **TO**  $n_1$  **DO**
6.      $L[i] = A[p + i - 1]$
7. **ENDDO**
8. **FOR**  $j = 1$  **TO**  $n_2$  **DO**
9.      $R[j] = A[q + j]$
10. **ENDDO**
11.  $L[n_1 + 1] = \infty$
12.  $R[n_2 + 1] = \infty$
13.  $i = 1$
14.  $j = 1$
15. **FOR**  $k = p$  **TO**  $r$  **DO**
16.      $\dots$
23. **ENDDO**

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

**Beispiel:**

*Merge*( $A, 9, 12, 16$ )

$A$	8	9	10	11	12	13	14	15	16	17	...
	...	2	4	5	7	1	2	3	6	...	

# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

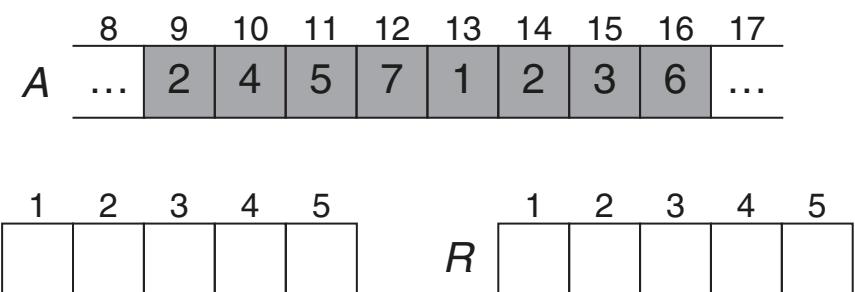
1.  $n_1 = q - p + 1$
2.  $n_2 = r - p$
3.  $L = \text{array}(n_1 + 1)$
4.  $R = \text{array}(n_2 + 1)$
5. **FOR**  $i = 1$  **TO**  $n_1$  **DO**
6.      $L[i] = A[p + i - 1]$
7. **ENDDO**
8. **FOR**  $j = 1$  **TO**  $n_2$  **DO**
9.      $R[j] = A[q + j]$
10. **ENDDO**
11.  $L[n_1 + 1] = \infty$
12.  $R[n_2 + 1] = \infty$
13.  $i = 1$
14.  $j = 1$
15. **FOR**  $k = p$  **TO**  $r$  **DO**
- |     ...
23. **ENDDO**

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

**Beispiel:**

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

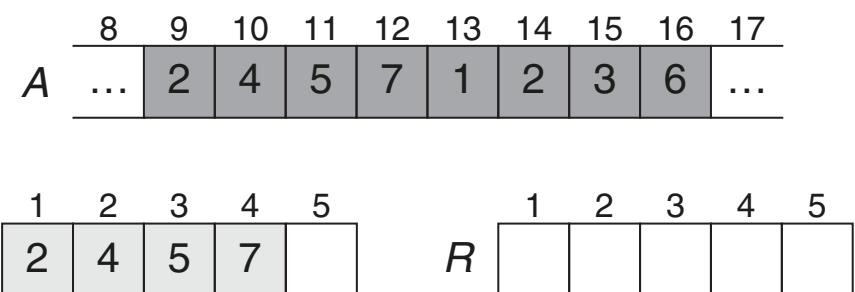
1.  $n_1 = q - p + 1$
2.  $n_2 = r - p$
3.  $L = \text{array}(n_1 + 1)$
4.  $R = \text{array}(n_2 + 1)$
5. **FOR**  $i = 1$  **TO**  $n_1$  **DO**
6.      $L[i] = A[p + i - 1]$
7. **ENDDO**
8. **FOR**  $j = 1$  **TO**  $n_2$  **DO**
9.      $R[j] = A[q + j]$
10. **ENDDO**
11.  $L[n_1 + 1] = \infty$
12.  $R[n_2 + 1] = \infty$
13.  $i = 1$
14.  $j = 1$
15. **FOR**  $k = p$  **TO**  $r$  **DO**
- |     ...
23. **ENDDO**

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

**Beispiel:**

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

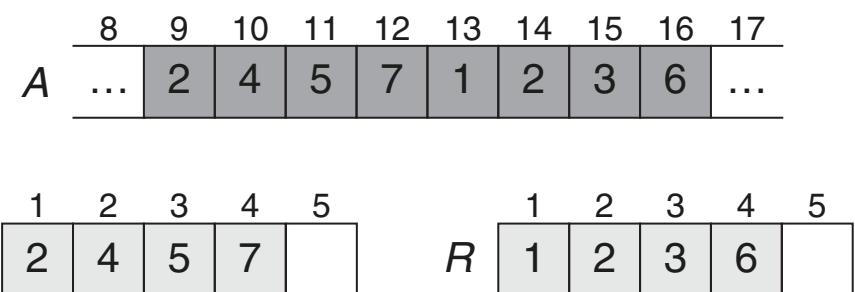
1.  $n_1 = q - p + 1$
2.  $n_2 = r - p$
3.  $L = \text{array}(n_1 + 1)$
4.  $R = \text{array}(n_2 + 1)$
5. **FOR**  $i = 1$  **TO**  $n_1$  **DO**
6.      $L[i] = A[p + i - 1]$
7. **ENDDO**
8. **FOR**  $j = 1$  **TO**  $n_2$  **DO**
9.      $R[j] = A[q + j]$
10. **ENDDO**
11.  $L[n_1 + 1] = \infty$
12.  $R[n_2 + 1] = \infty$
13.  $i = 1$
14.  $j = 1$
15. **FOR**  $k = p$  **TO**  $r$  **DO**
- |     ...
23. **ENDDO**

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

**Beispiel:**

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

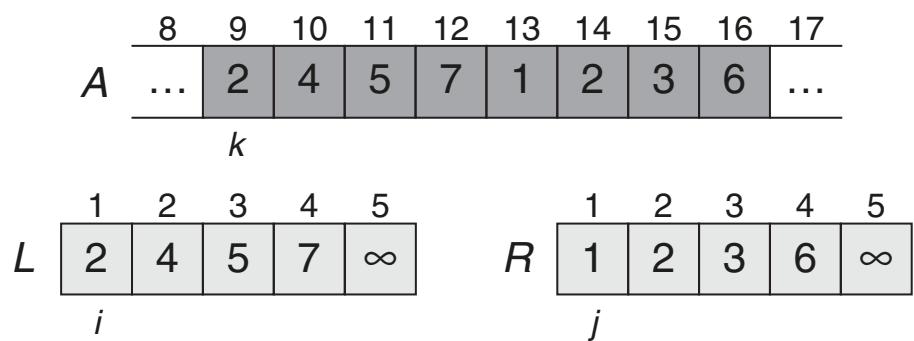
1.  $n_1 = q - p + 1$
2.  $n_2 = r - p$
3.  $L = \text{array}(n_1 + 1)$
4.  $R = \text{array}(n_2 + 1)$
5. **FOR**  $i = 1$  **TO**  $n_1$  **DO**
6.      $L[i] = A[p + i - 1]$
7. **ENDDO**
8. **FOR**  $j = 1$  **TO**  $n_2$  **DO**
9.      $R[j] = A[q + j]$
10. **ENDDO**
11.  $L[n_1 + 1] = \infty$
12.  $R[n_2 + 1] = \infty$
13.  $i = 1$
14.  $j = 1$
15. **FOR**  $k = p$  **TO**  $r$  **DO**
- |     ...
23. **ENDDO**

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

**Beispiel:**

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

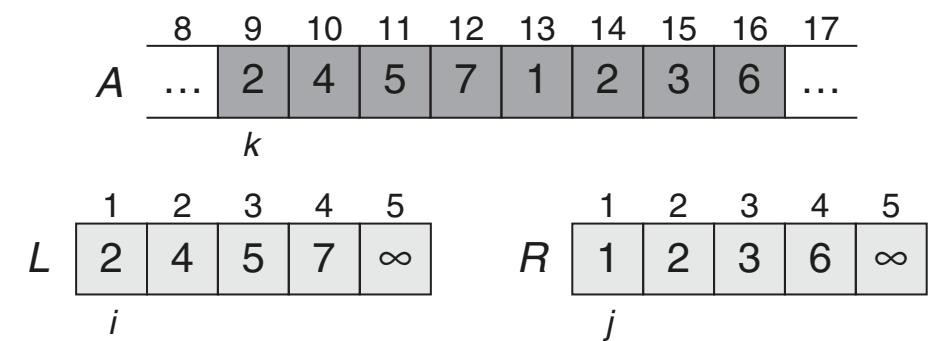
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

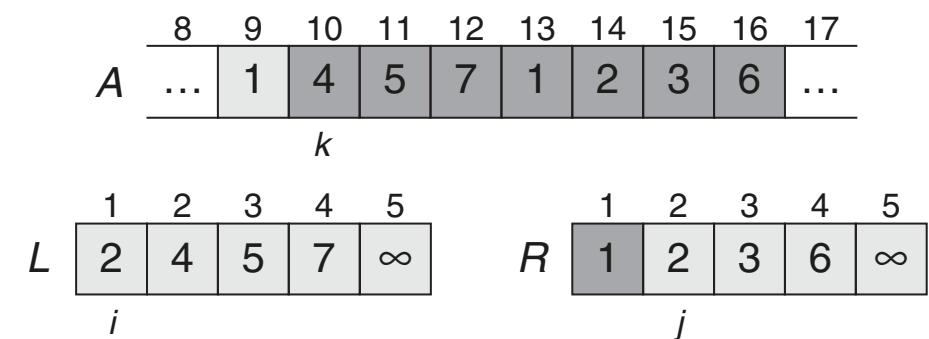
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

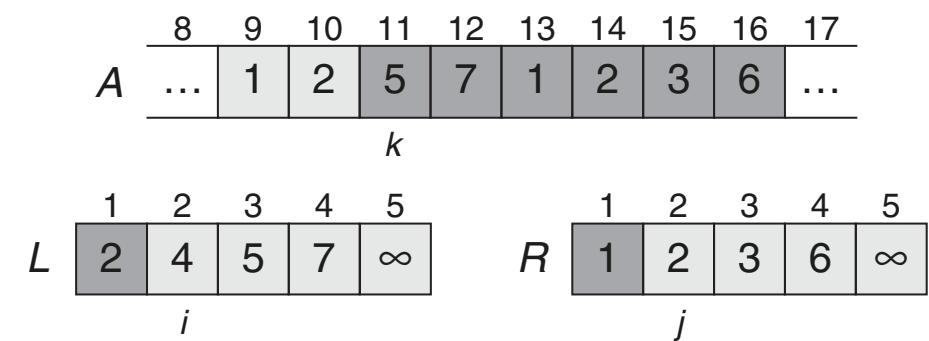
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

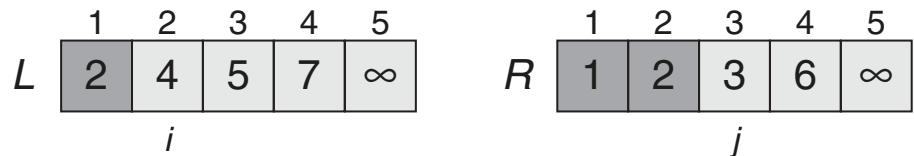
Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )

$A$	8	9	10	11	12	13	14	15	16	17	$k$
	...	1	2	2	7	1	2	3	6	...	



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

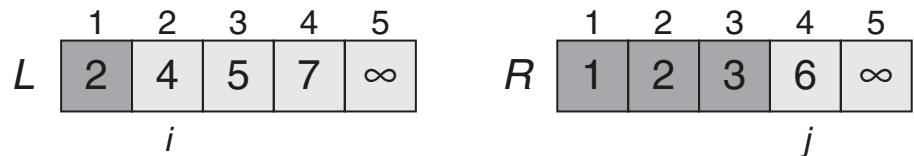
Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )

A	8	9	10	11	12	13	14	15	16	17	
	...	1	2	2	3	1	2	3	6	...	



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

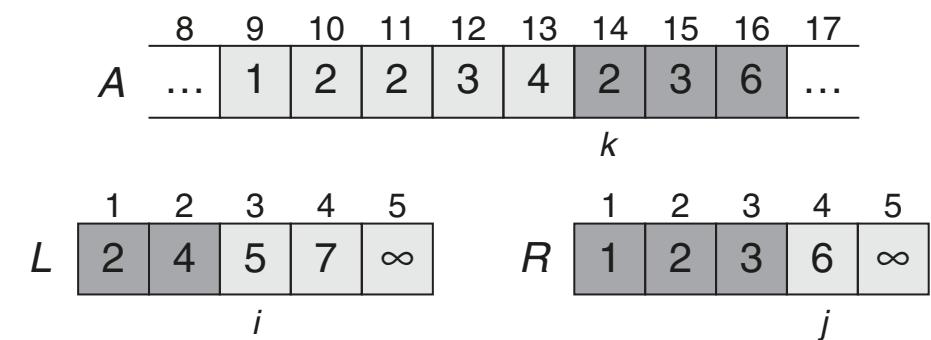
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

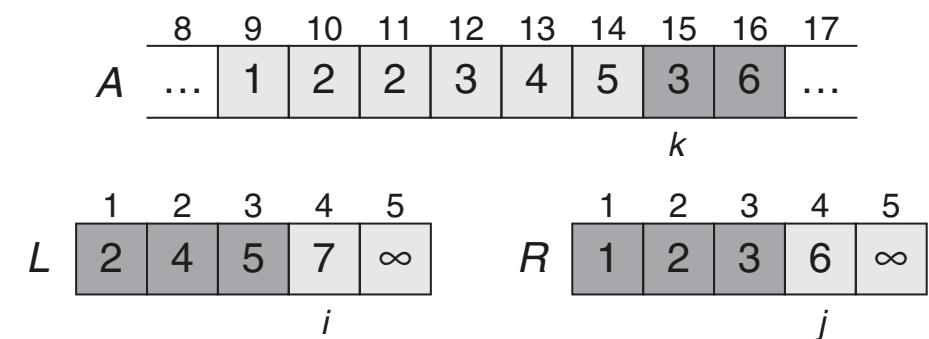
```
1.   $n_1 = q - p + 1$ 
|  ...
13.  $i = 1$ 
14.  $j = 1$ 
15. FOR  $k = p$  TO  $r$  DO
16.   IF  $L[i] \leq R[j]$  THEN
17.      $A[k] = L[i]$ 
18.      $i = i + 1$ 
19.   ELSE
20.      $A[k] = R[j]$ 
21.      $j = j + 1$ 
22.   ENDIF
23. ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )



# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )

$A$	8	9	10	11	12	13	14	15	16	17	...
	...										$k$

$L$	1	2	3	4	5	$i$
						$\infty$

$R$	1	2	3	4	5	$j$
						$\infty$

# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen  
Hinzufügen von  $\infty$  am Ende von  $L$  und  $R$  vermeidet wiederholtes Prüfen auf Überschreitung von  $n_1$  und  $n_2$  durch  $i$  und  $j$ .

Beispiel:

*Merge*( $A, 9, 12, 16$ )

$A$	8	9	10	11	12	13	14	15	16	17	$\dots$
	...									$k$	

$L$	1	2	3	4	5	$\infty$
	2	4	5	7	$\infty$	

$R$	1	2	3	4	5	$\infty$
	1	2	3	6	$\infty$	

# Merge Sort

## Merge

*Merge*( $A, p, q, r$ )

1.  $n_1 = q - p + 1$
2.  $n_2 = r - p$
3.  $L = \text{array}(n_1 + 1)$
4.  $R = \text{array}(n_2 + 1)$
5. **FOR**  $i = 1$  **TO**  $n_1$  **DO**
6.      $L[i] = A[p + i - 1]$
7. **ENDDO**
8. **FOR**  $j = 1$  **TO**  $n_2$  **DO**
9.      $R[j] = A[q + j]$
10. **ENDDO**
11.  $L[n_1 + 1] = \infty$
12.  $R[n_2 + 1] = \infty$
13.  $i = 1$
14.  $j = 1$
15. **FOR**  $k = p$  **TO**  $r$  **DO**  
|     ...
23. **ENDDO**

Laufzeit:

- $n = r - p + 1$  ist die Summe der Längen der Teilarrays.
- For-Schleifen zum Kopieren:  
 $\Theta(n_1 + n_2) = \Theta(n)$  Zeit.
- For-Schleife zum Vereinen:  
 $n \cdot \Theta(1) = \Theta(n)$
- Gesamlaufzeit:  $\Theta(n)$ .

# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*( $A, p, q$ )
4.     *MergeSort*( $A, q + 1, r$ )
5.     *Merge*( $A, p, q, r$ )
6. **ENDIF**

# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**

2.    $q = \lfloor (p + r)/2 \rfloor$

3.   *MergeSort*( $A, p, q$ )

4.   *MergeSort*( $A, q + 1, r$ )

5.   *Merge*( $A, p, q, r$ )

6. **ENDIF**

Laufzeitfunktion:

$$T(n) = c_1$$

$$+ D(n)$$

$$+ T(n/2)$$

$$+ T(n/2)$$

$$+ C(n)$$

$$+ c_6$$

# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*( $A, p, q$ )
4.     *MergeSort*( $A, q + 1, r$ )
5.     *Merge*( $A, p, q, r$ )
6. **ENDIF**

Laufzeitfunktion:

$$\begin{aligned} T(n) &= \Theta(1) \\ &+ \Theta(1) \\ &+ T(n/2) \\ &+ T(n/2) \\ &+ C(n) \\ &+ \Theta(1) \end{aligned}$$

# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*( $A, p, q$ )
4.     *MergeSort*( $A, q + 1, r$ )
5.     *Merge*( $A, p, q, r$ )
6. **ENDIF**

Laufzeitfunktion:

$$\begin{aligned} T(n) &= \Theta(1) \\ &+ \Theta(1) \\ &+ T(n/2) \\ &+ T(n/2) \\ &+ \Theta(n) \\ &+ \Theta(1) \end{aligned}$$

# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*( $A, p, q$ )
4.     *MergeSort*( $A, q + 1, r$ )
5.     *Merge*( $A, p, q, r$ )
6. **ENDIF**

Laufzeitfunktion:

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1, \\ 2 \cdot T(n/2) + \Theta(n) & \text{für } n > 1. \end{cases}$$

# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*( $A, p, q$ )
4.     *MergeSort*( $A, q + 1, r$ )
5.     *Merge*( $A, p, q, r$ )
6. **ENDIF**

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

# Merge Sort

## Laufzeitanalyse

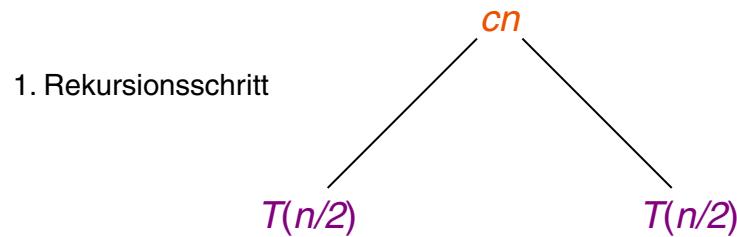
*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**
2.      $q = \lfloor (p + r)/2 \rfloor$
3.     *MergeSort*( $A, p, q$ )
4.     *MergeSort*( $A, q + 1, r$ )
5.     *Merge*( $A, p, q, r$ )
6. **ENDIF**

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

Rekursionsbaum:



# Merge Sort

## Laufzeitanalyse

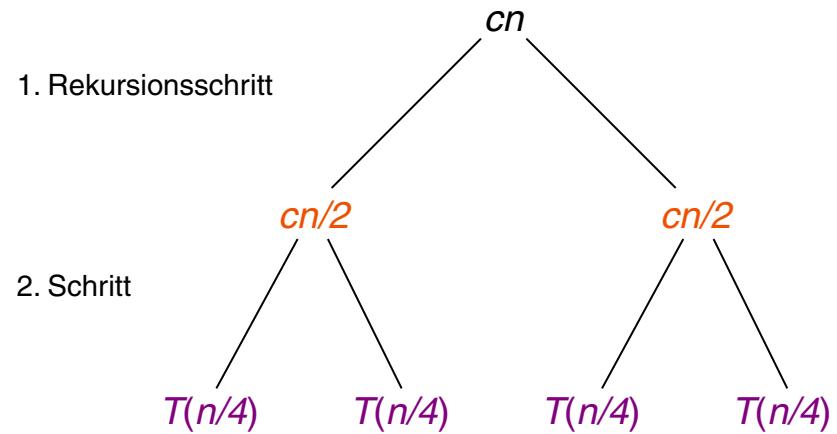
*MergeSort*( $A, p, r$ )

1. IF  $p < r$  THEN
2.  $q = \lfloor (p + r)/2 \rfloor$
3. *MergeSort*( $A, p, q$ )
4. *MergeSort*( $A, q + 1, r$ )
5. *Merge*( $A, p, q, r$ )
6. ENDIF

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

Rekursionsbaum:



# Merge Sort

## Laufzeitanalyse

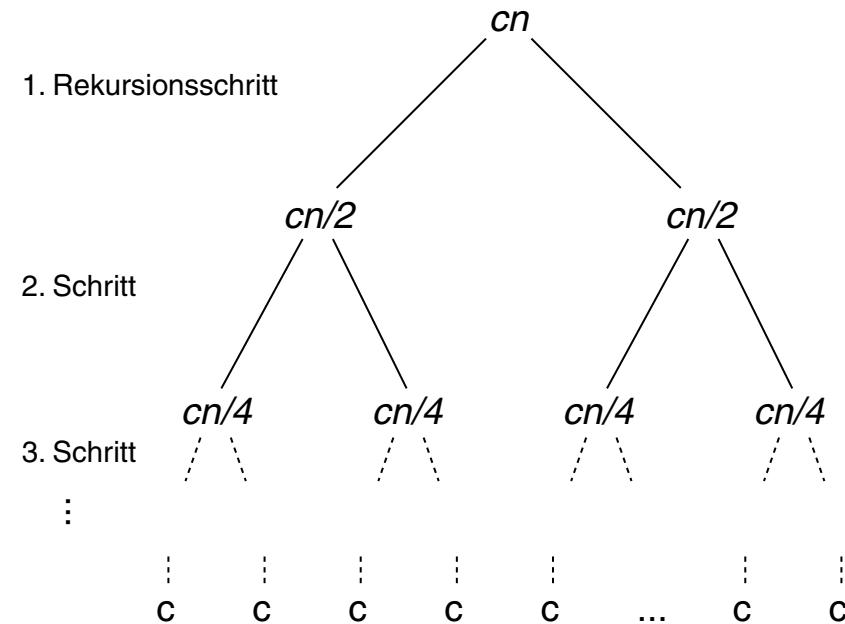
*MergeSort*( $A, p, r$ )

1. IF  $p < r$  THEN
2.  $q = \lfloor (p + r)/2 \rfloor$
3. *MergeSort*( $A, p, q$ )
4. *MergeSort*( $A, q + 1, r$ )
5. *Merge*( $A, p, q, r$ )
6. ENDIF

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

Rekursionsbaum:



# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. **IF**  $p < r$  **THEN**
2.    $q = \lfloor (p+r)/2 \rfloor$
3.   *MergeSort*( $A, p, q$ )
4.   *MergeSort*( $A, q+1, r$ )
5.   *Merge*( $A, p, q, r$ )
6. **ENDIF**

Vereinfachende Annahme:

- $n$  ist 2er-Potenz.

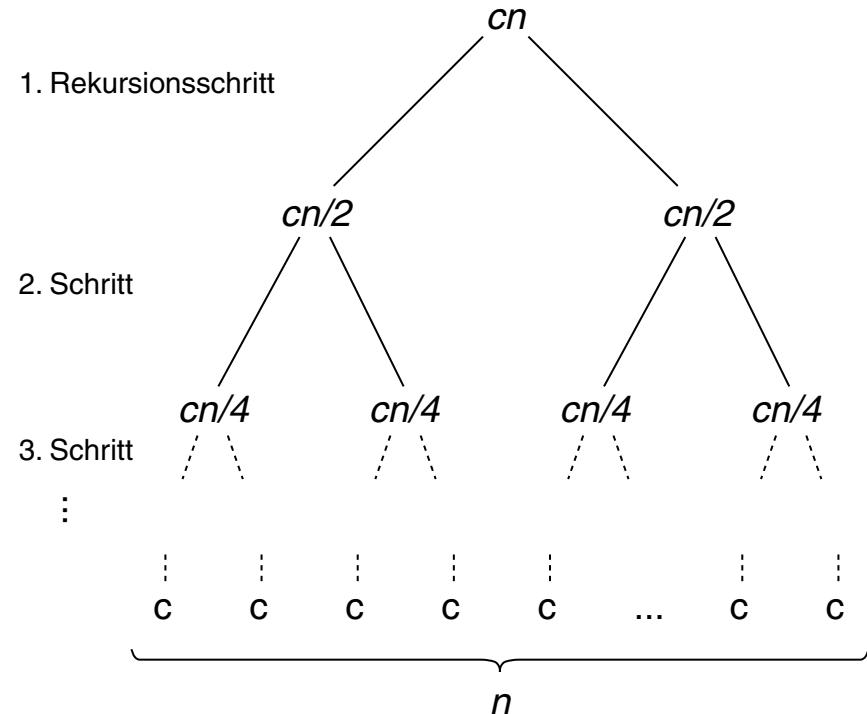
Worst Case:

- Unterste Ebene vollständig gefüllt.

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

Rekursionsbaum:



# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. IF  $p < r$  THEN
2.  $q = \lfloor (p+r)/2 \rfloor$
3. *MergeSort*( $A, p, q$ )
4. *MergeSort*( $A, q+1, r$ )
5. *Merge*( $A, p, q, r$ )
6. ENDIF

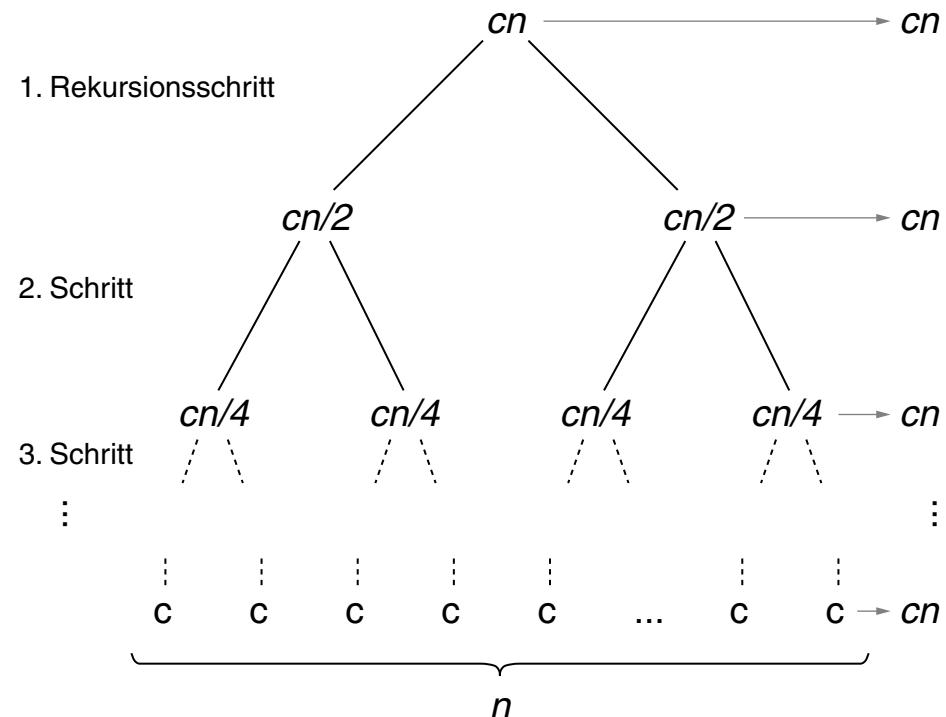
Beobachtungen:

- Kosten pro Ebene:  $n$

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

Rekursionsbaum:



# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. IF  $p < r$  THEN
2.  $q = \lfloor (p+r)/2 \rfloor$
3. *MergeSort*( $A, p, q$ )
4. *MergeSort*( $A, q+1, r$ )
5. *Merge*( $A, p, q, r$ )
6. ENDIF

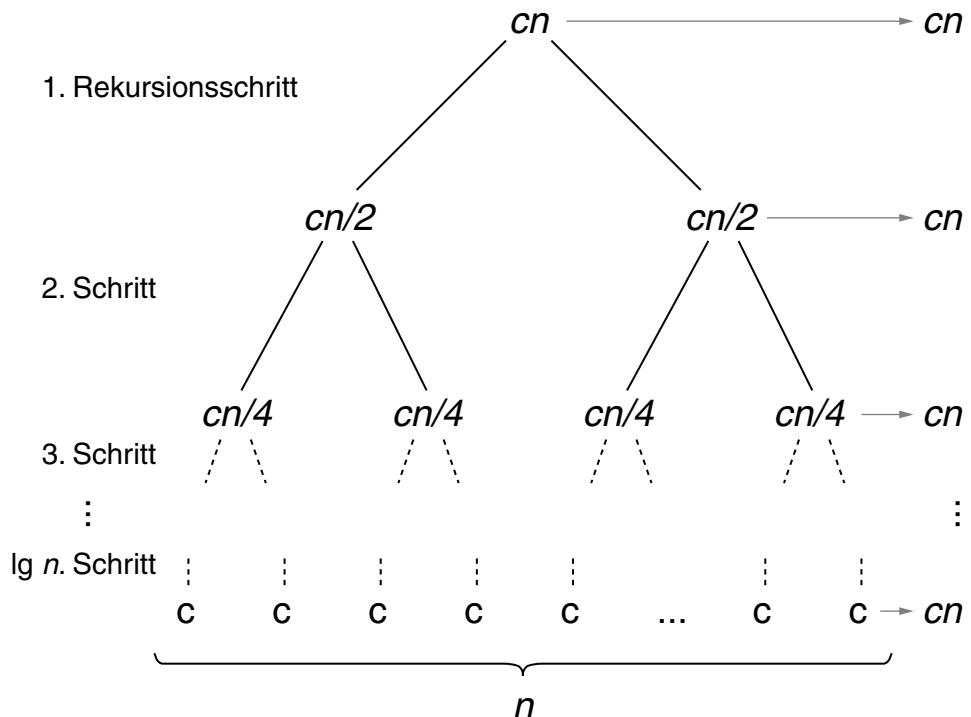
Beobachtungen:

- Kosten pro Ebene:  $n$
- Anzahl Schritte:  $\lg n$
- Anzahl Ebenen:  $\lg n + 1$

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

Rekursionsbaum:



# Merge Sort

## Laufzeitanalyse

*MergeSort*( $A, p, r$ )

1. IF  $p < r$  THEN
2.  $q = \lfloor (p+r)/2 \rfloor$
3. *MergeSort*( $A, p, q$ )
4. *MergeSort*( $A, q+1, r$ )
5. *Merge*( $A, p, q, r$ )
6. ENDIF

Beobachtungen:

- Kosten pro Ebene:  $n$
- Anzahl Schritte:  $\lg n$
- Anzahl Ebenen:  $\lg n + 1$

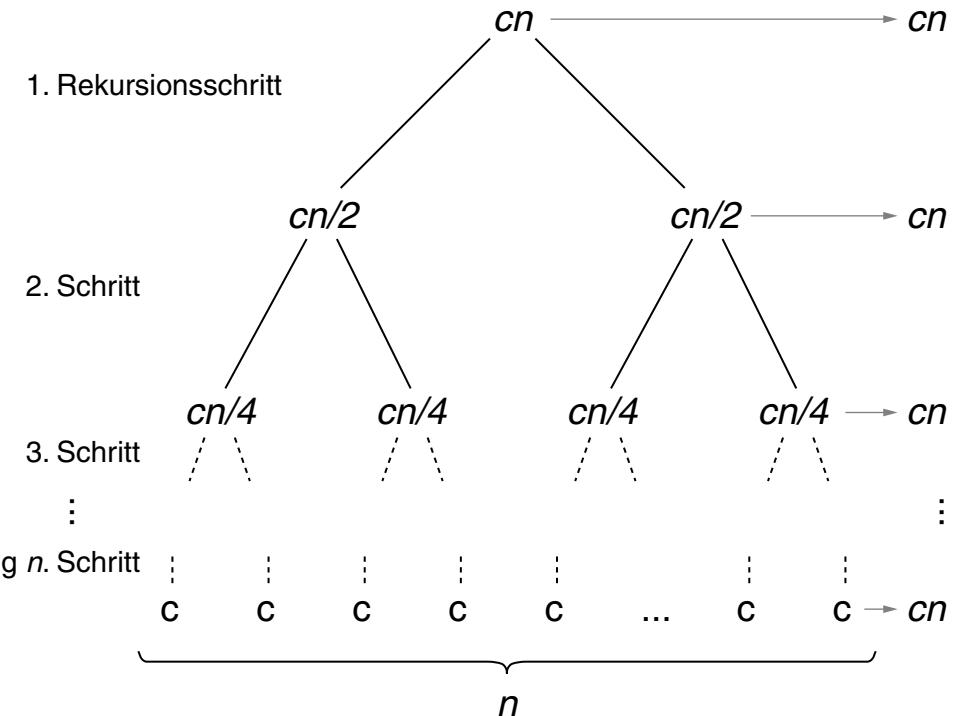
Laufzeitkomplexität:

- $T(n) = (\lg n + 1) \cdot n$
- $T(n) = \Theta(n \lg n)$  ?

Laufzeitfunktion (vereinfacht):

$$T(n) = \begin{cases} 1 & \text{für } n = 1, \\ 2 \cdot T(n/2) + n & \text{für } n > 1. \end{cases}$$

Rekursionsbaum:



## Bemerkungen:

- Abschätzung der Baumtiefe: Die Größe des Teilproblem auf Ebene  $i$  entspricht  $n/2^i$ . Der Basisfall  $n = 1$  ist genau dann erreicht, wenn  $n/2^i = 1$ , also  $i = \log_2 n$ .
- Abkürzend schreiben wir  $\log_2 n = \lg n$ .
- Bei besonderer Sorgfalt kann die Rekursionsbaummethode als Beweis dienen. Aussagekräftiger im Allgemeinen ist jedoch die Substitutionsmethode.

# Heapsort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortieren durch Selektion

# Heapsort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

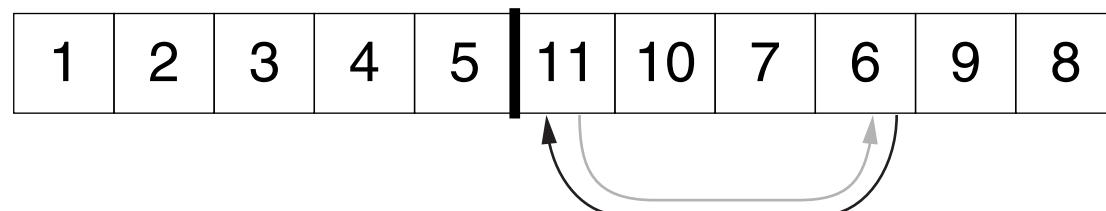
Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortieren durch Selektion

Naives Selektionsverfahren: (wie bei Selection Sort)

- Sequentielle Suche des nächstgrößten Elements unter den unsortierten.
- Quadratische Laufzeit.

Beispiel:



# Heapsort

## Algorithmus

Problem: Sortieren

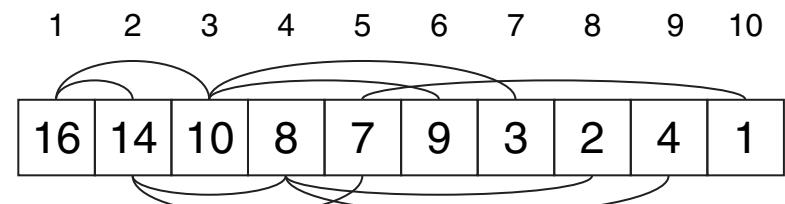
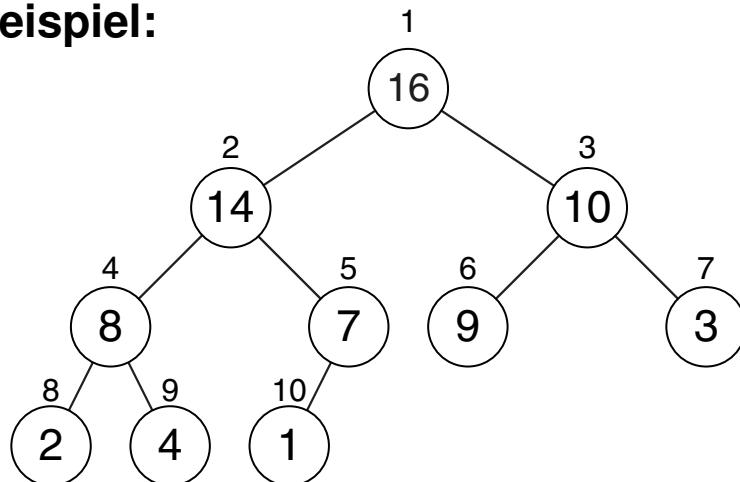
Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortieren durch Selektion unter Verwendung eines **Heaps**.

Beispiel:



# Einschub: Binary Heap

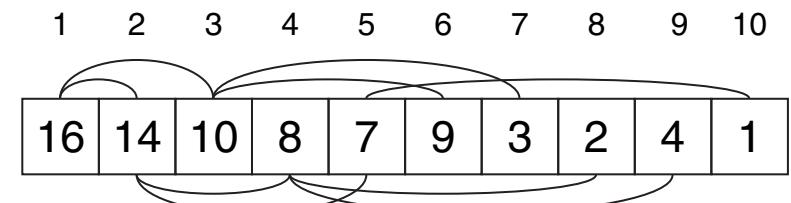
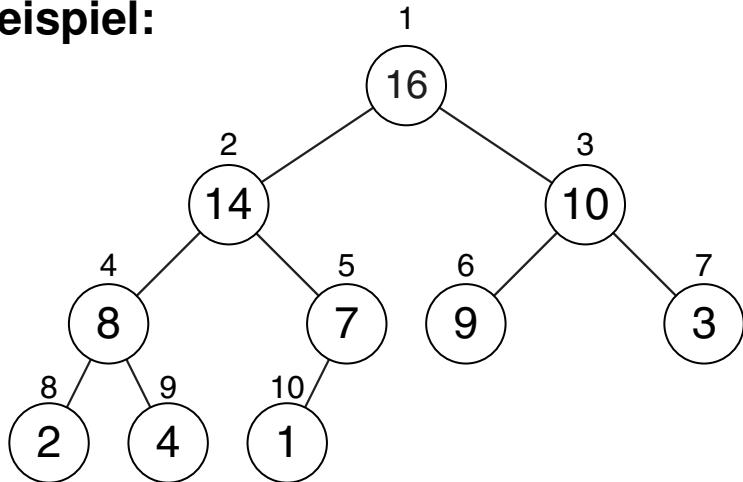
## Definition

Ein vollständiger Binärbaum  $A$  heißt Max-Heap (Min-Heap), wenn jeder Knoten  $i$  mit Ausnahme der Wurzel folgende Bedingung erfüllt:

$$A[i] \leq A[\text{parent}(i)] \quad (A[i] \geq A[\text{parent}(i)]),$$

wobei  $A[i]$  Sortierschlüssel von  $i$  ist und  $\text{parent}(i)$  den Elternknoten von  $i$  ergibt.

## Beispiel:



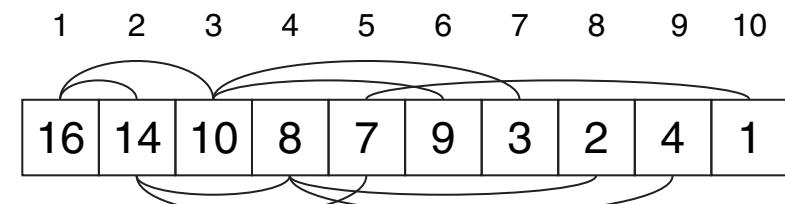
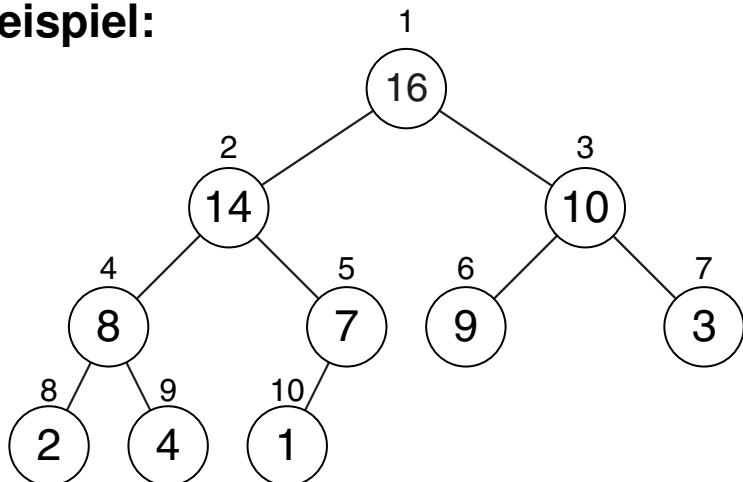
# Einschub: Binary Heap

## Implementierung

Speicherrepräsentation:

- Ein Heap mit  $n$  Knoten wird in einem Array  $A$  der Länge  $m \geq n$  gespeichert.
- Die Heap-Größe  $n$  kann im Intervall  $[0, m]$  verändert werden.
- Ebenen werden der Reihe nach von  $j = 0$  bis  $j = \lfloor \lg n \rfloor$  in  $A$  gespeichert.
- Ebene  $j$  benötigt  $2^j$  Speicherplätze, Ebene  $\lfloor \lg n \rfloor$  benötigt  $n - 2^{\lfloor \lg n \rfloor} + 1$ .

Beispiel:



# Einschub: Binary Heap

## Implementierung

### Hilfsfunktionen

Eingabe:   *i.* Index eines Knotens eines Binärbaums repräsentiert als Array.

Ausgabe:   Index eines verwandten Knotens.

*parent(i)*

1. *return*( $\lfloor i/2 \rfloor$ )

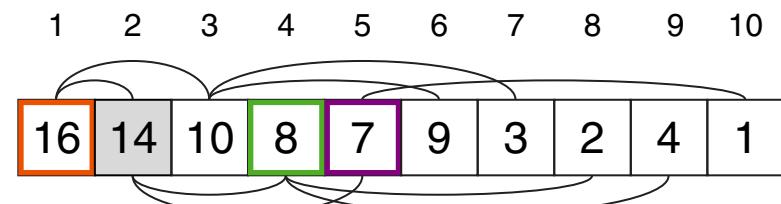
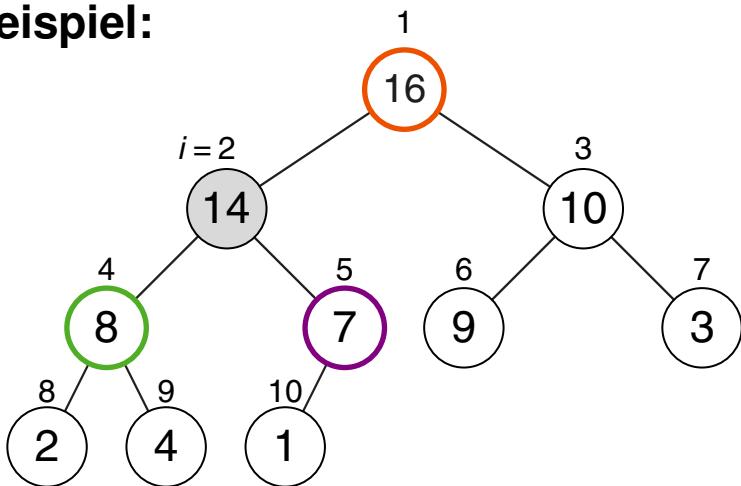
*leftChild(i)*

1. *return*( $2i$ )

*rightChild(i)*

1. *return*( $2i + 1$ )

### Beispiel:



## Bemerkungen:

- ❑ Die Bedingung, die ein Heap erfüllen muss, wird auch „Heap-Bedingung“ genannt.
- ❑ Ein vollständiger Binärbaum ist ein Binärbaum, bei dem alle Ebenen außer der untersten voll ausgefüllt sind. Auf der untersten Ebene sind alle Knoten so weit links, wie möglich.
- ❑ Die Implementierung von Hilfsfunktionen wird mit Bit-Shift-Instruktionen umgesetzt.

# Einschub: Binary Heap

## Konstruktion

### Algorithmen:

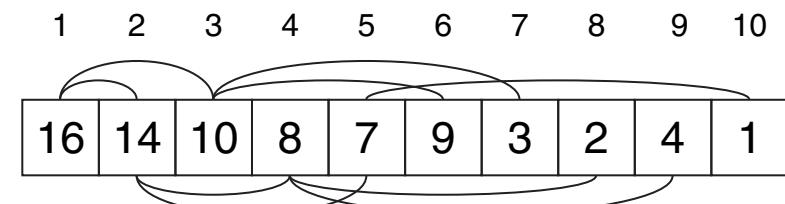
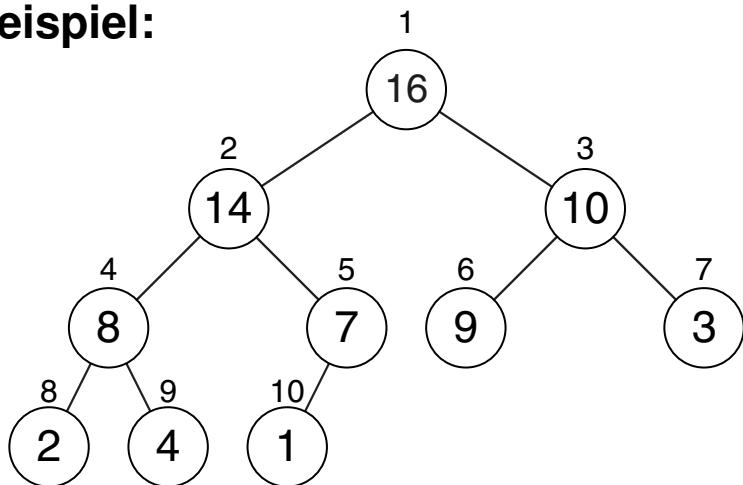
- **Max-Heapify (Min-Heapify)**

Rekursive Herstellung der Heap-Bedingung für einen gegebenen Knoten eines Binärbaums unter der Voraussetzung, dass sein linker und rechter Teilbaum jeweils Heaps sind.

- **Build-Max-Heap (Build-Min-Heap)**

Iterative Herstellung der Heap-Bedingung für alle Knoten eines Binärbaums mittels Max-Heapify (Min-Heapify).

### Beispiel:



# Einschub: Binary Heap

## Konstruktion

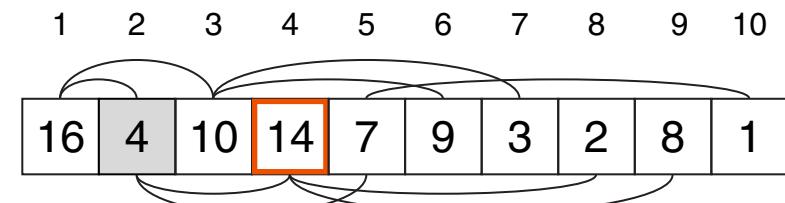
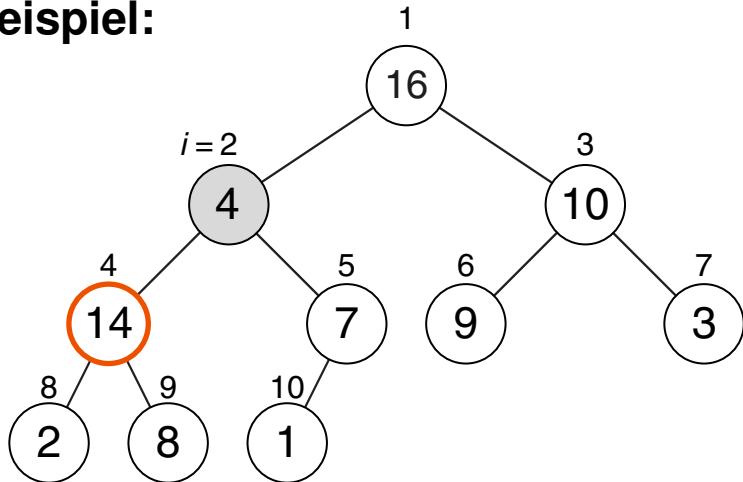
Algorithmus: Max-Heapify

Eingabe: A. Binärbaum mit  $n$  Knoten als Array der Länge  $m \geq n$ .

i. Knoten des Binärbaums.

Ausgabe: Binärbaum dessen Teilbaum mit Wurzel  $i$  ein Heap ist,  
falls der linke und rechte Teilbaum von  $i$  jeweils Heaps waren.

Beispiel:



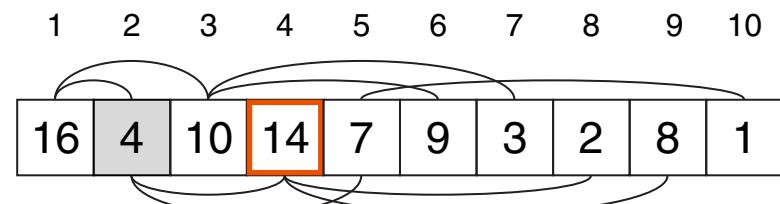
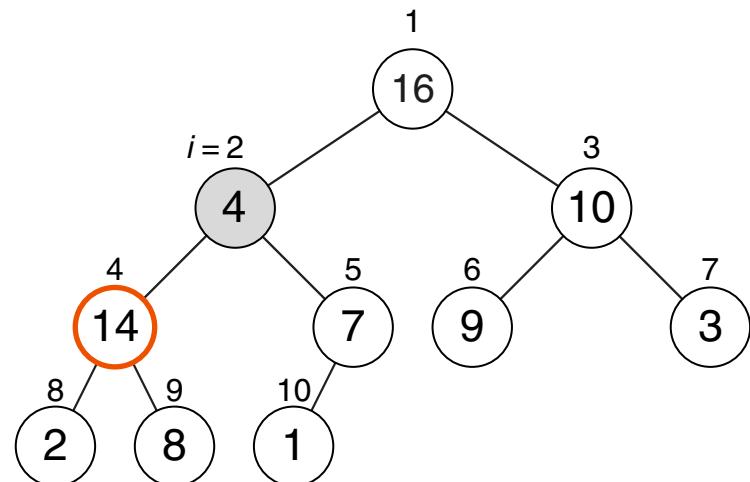
# Einschub: Binary Heap

## Konstruktion

*MaxHeapify(A, i)*

1.  $l = \text{leftChild}(i)$
2.  $r = \text{rightChild}(i)$
3. **IF**  $l \leq n$  **AND**  $A[l] > A[i]$  **THEN**
4.     *largest* =  $l$
5. **ELSE**
6.     *largest* =  $i$
7. **ENDIF**
8. **IF**  $r \leq n$  **AND**  $A[r] > A[\text{largest}]$  **THEN**
9.     *largest* =  $r$
10. **ENDIF**
11. **IF**  $\text{largest} \neq i$  **THEN**
12.      $a_i = A[i]$
13.      $A[i] = A[\text{largest}]$
14.      $A[\text{largest}] = a_i$
15.     *MaxHeapify(A, largest)*
16. **ENDIF**

**Beispiel:**



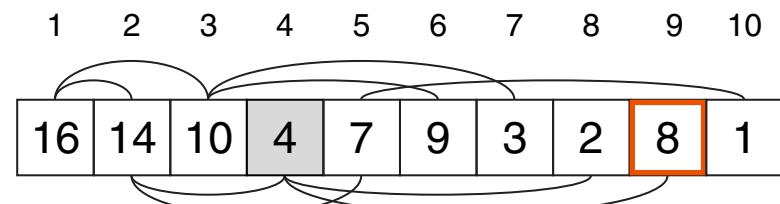
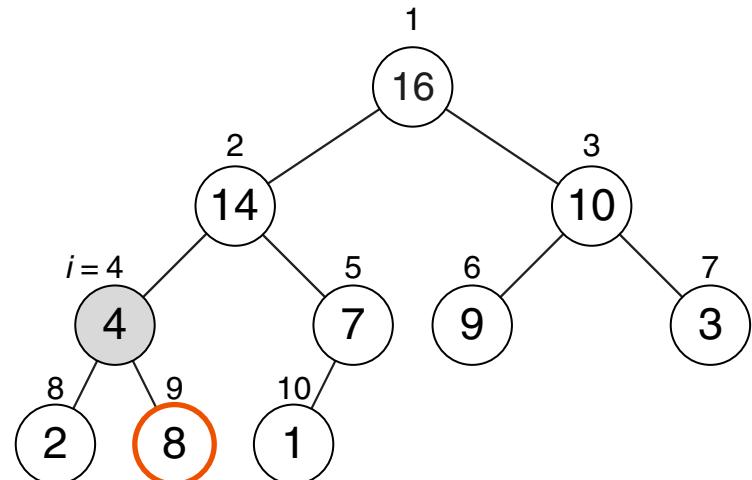
# Einschub: Binary Heap

## Konstruktion

*MaxHeapify(A, i)*

1.  $l = \text{leftChild}(i)$
2.  $r = \text{rightChild}(i)$
3. **IF**  $l \leq n$  **AND**  $A[l] > A[i]$  **THEN**
4.      $\text{largest} = l$
5. **ELSE**
6.      $\text{largest} = i$
7. **ENDIF**
8. **IF**  $r \leq n$  **AND**  $A[r] > A[\text{largest}]$  **THEN**
9.      $\text{largest} = r$
10. **ENDIF**
11. **IF**  $\text{largest} \neq i$  **THEN**
12.      $a_i = A[i]$
13.      $A[i] = A[\text{largest}]$
14.      $A[\text{largest}] = a_i$
15.     *MaxHeapify(A, largest)*
16. **ENDIF**

**Beispiel:**



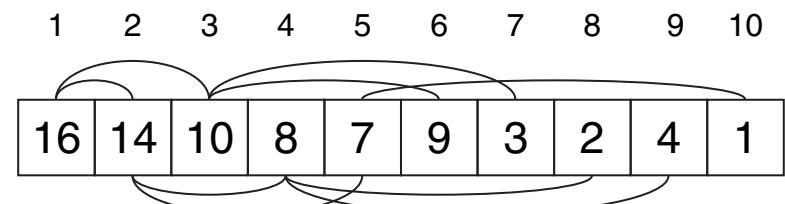
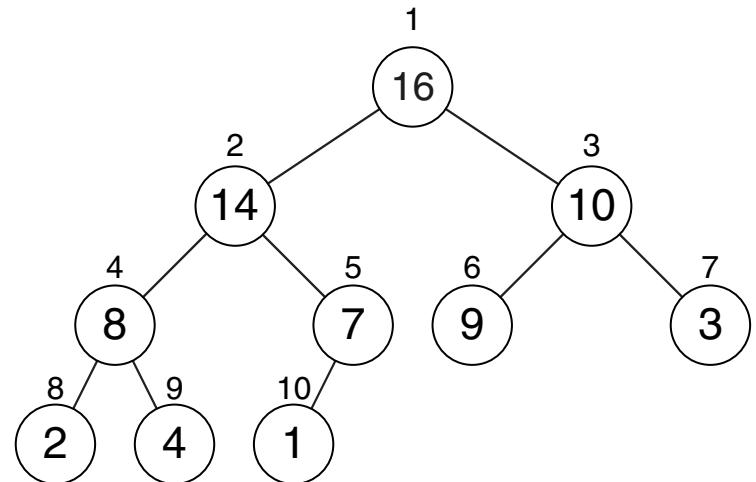
# Einschub: Binary Heap

## Konstruktion

*MaxHeapify(A, i)*

1.  $l = \text{leftChild}(i)$
2.  $r = \text{rightChild}(i)$
3. **IF**  $l \leq n$  **AND**  $A[l] > A[i]$  **THEN**
4.      $\text{largest} = l$
5. **ELSE**
6.      $\text{largest} = i$
7. **ENDIF**
8. **IF**  $r \leq n$  **AND**  $A[r] > A[\text{largest}]$  **THEN**
9.      $\text{largest} = r$
10. **ENDIF**
11. **IF**  $\text{largest} \neq i$  **THEN**
12.      $a_i = A[i]$
13.      $A[i] = A[\text{largest}]$
14.      $A[\text{largest}] = a_i$
15.     *MaxHeapify(A, largest)*
16. **ENDIF**

**Beispiel:**



# Einschub: Binary Heap

## Konstruktion

*MaxHeapify(A, i)*

1.  $l = \text{leftChild}(i)$
2.  $r = \text{rightChild}(i)$
3. **IF**  $l \leq n$  **AND**  $A[l] > A[i]$  **THEN**
4.     *largest* =  $l$
5. **ELSE**
6.     *largest* =  $i$
7. **ENDIF**
8. **IF**  $r \leq n$  **AND**  $A[r] > A[\text{largest}]$  **THEN**
9.     *largest* =  $r$
10. **ENDIF**
11. **IF**  $\text{largest} \neq i$  **THEN**
12.      $a_i = A[i]$
13.      $A[i] = A[\text{largest}]$
14.      $A[\text{largest}] = a_i$
15.     *MaxHeapify(A, largest)*
16. **ENDIF**

Laufzeit:

- Zeilen 1-14 und 16:  $\Theta(1)$
- Zeile 15 (Worst Case):  
Teilbaum mit bis zu  $2n/3$  Knoten
  - $T(n) \leq T(2n/3) + \Theta(1)$
  - $T(n) = O(\lg n)$   
(gemäß Fall 2 des Master-Theorems)

# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

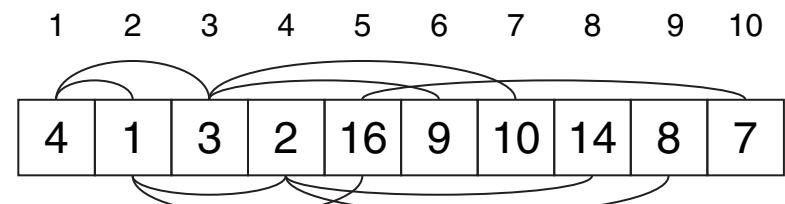
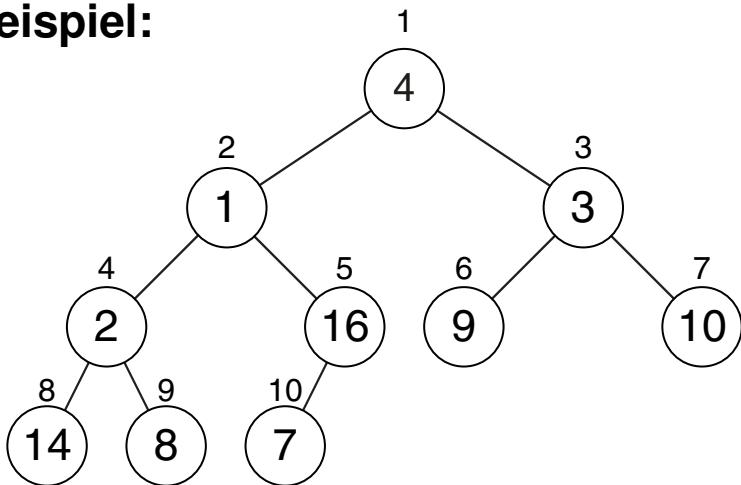
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.     *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

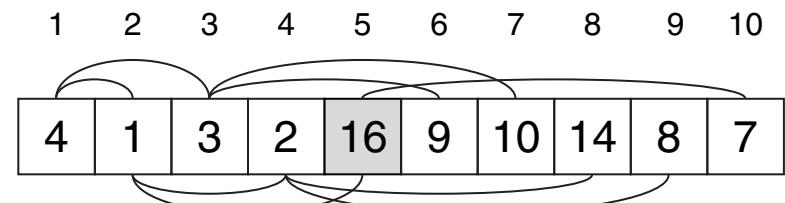
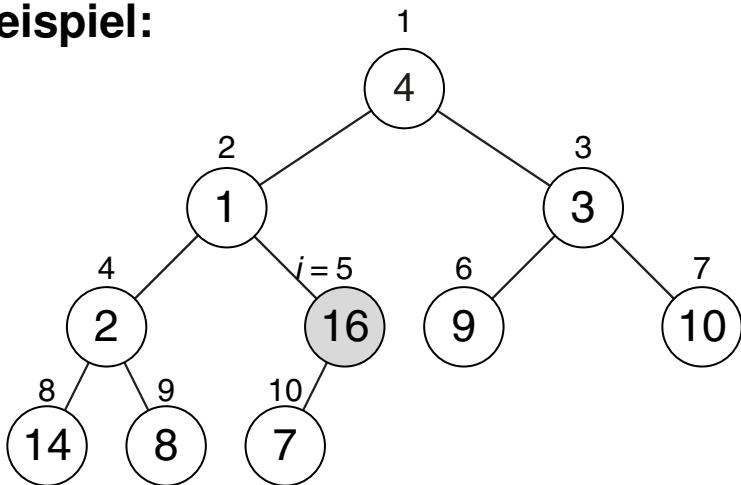
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.     *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

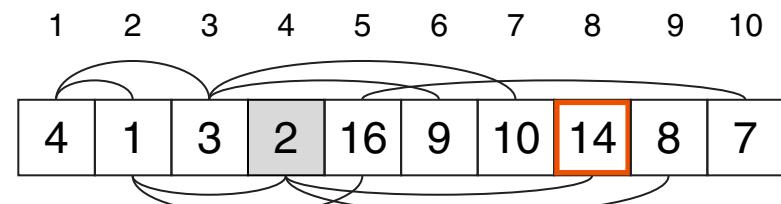
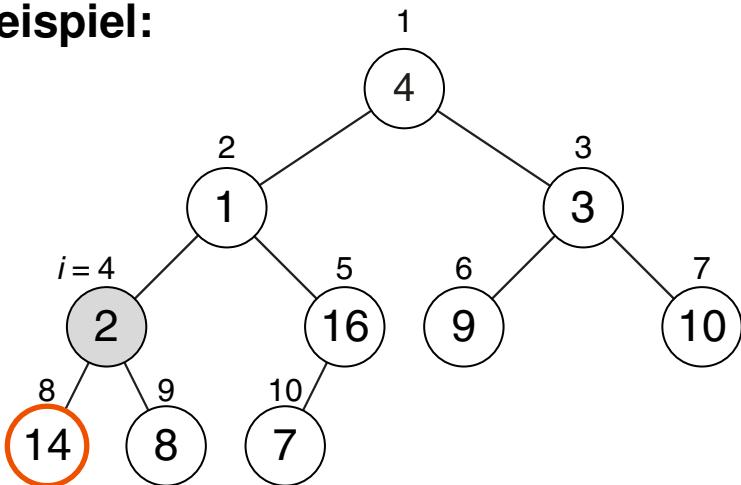
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.     *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

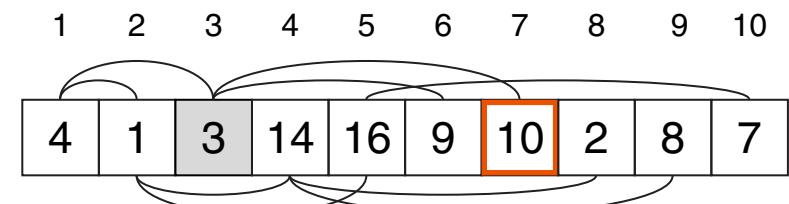
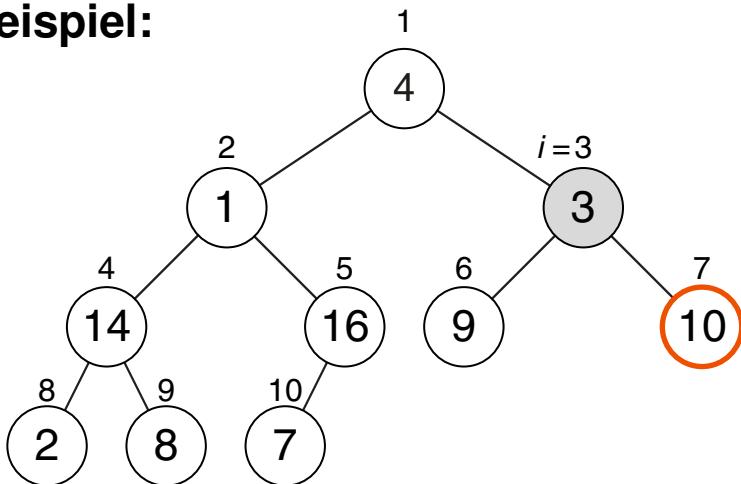
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.     *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

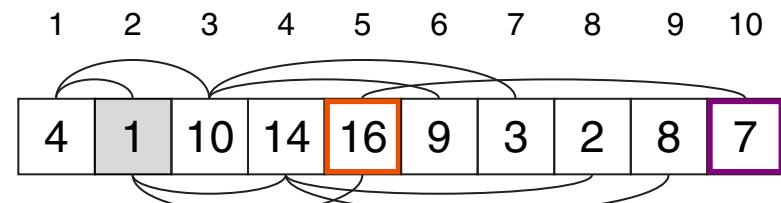
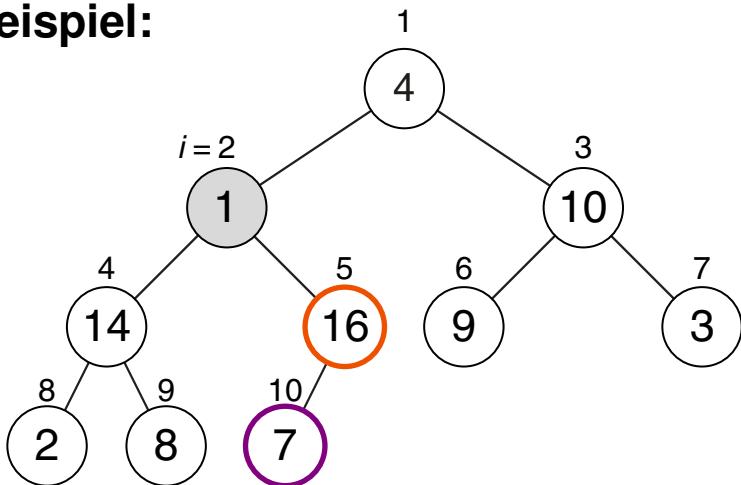
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.     *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

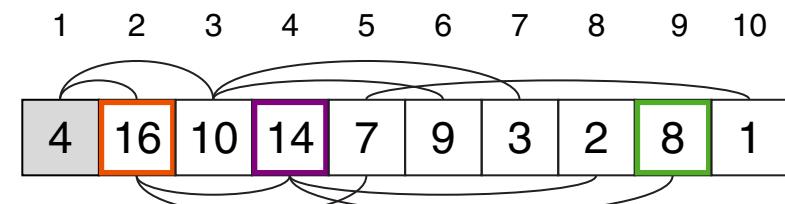
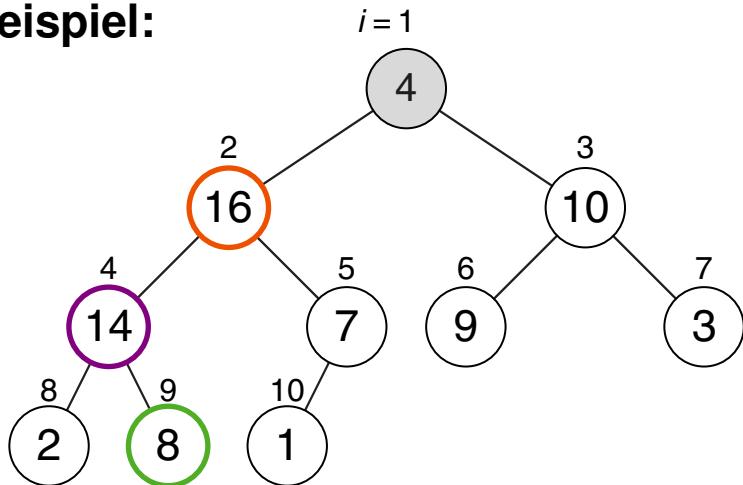
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.     *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

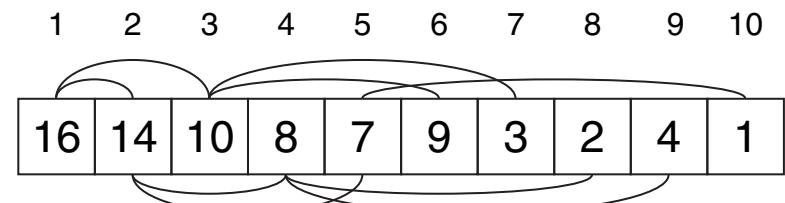
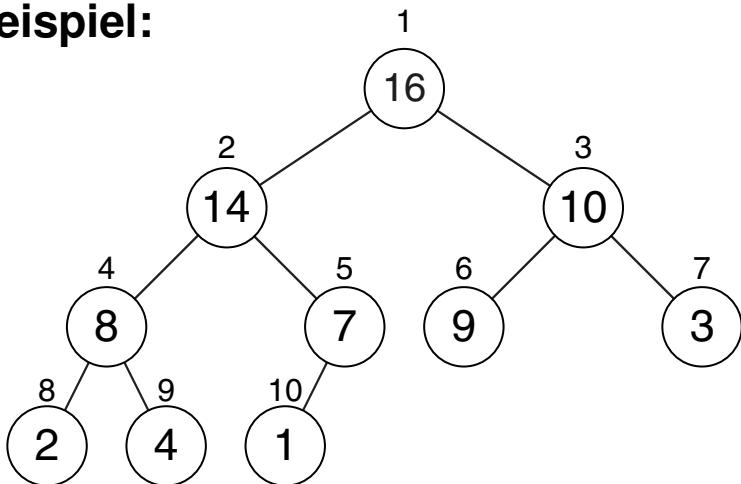
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.     *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



# Einschub: Binary Heap

## Konstruktion

Algorithmus: Build-Max-Heap

Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

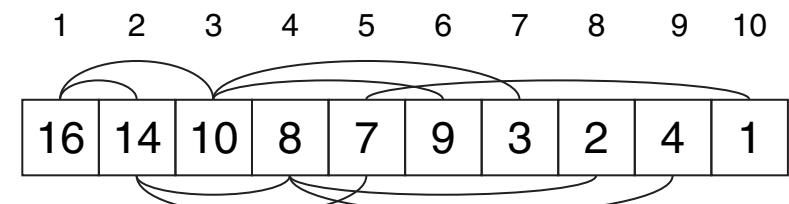
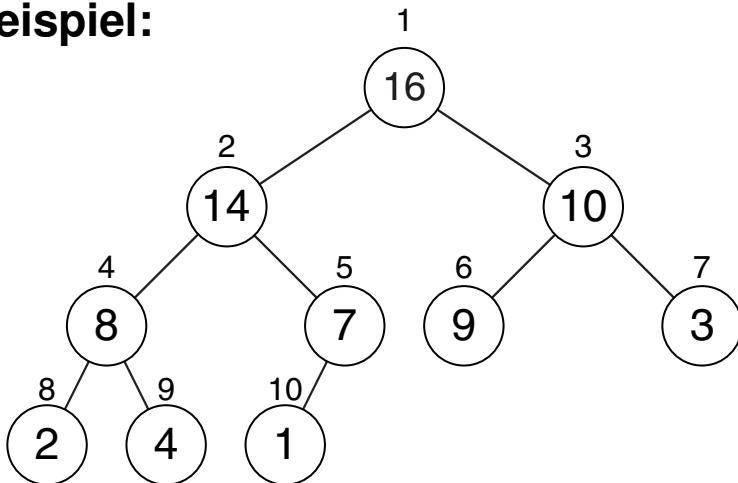
Laufzeit:

- $O(n)$  Max-Heapify-Aufrufe:  $O(n \lg n)$ .
- Starke Überschätzung.

*BuildMaxHeap(A)*

```
1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO  
2.   MaxHeapify( $A, i$ )  
3. ENDDO
```

Beispiel:



# Einschub: Binary Heap

## Konstruktion

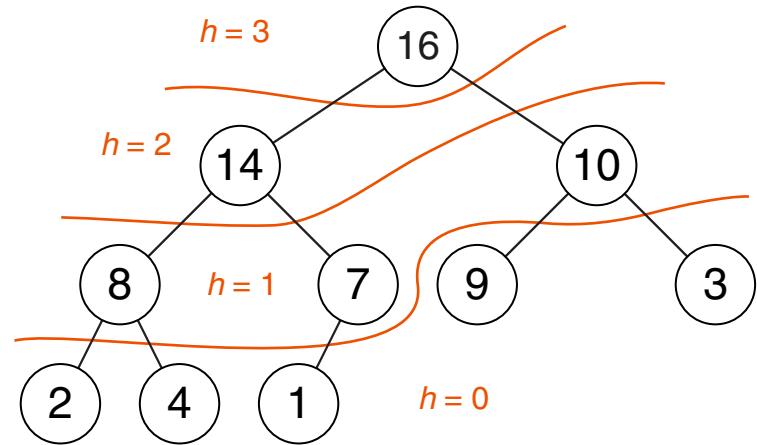
Algorithmus: Build-Max-Heap

Eingabe: A. Array der Länge  $n$ .

Ausgabe: Heap mit  $n$  Knoten.

*BuildMaxHeap(A)*

1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO
2.    *MaxHeapify(A, i)*
3. ENDDO



Laufzeitanalyse:

- Höhe  $h$  eines Knotens  $i$ : Längster direkter Pfad zu einem Blattknoten.
- Max-Heapify benötigt  $O(h)$  Zeit; die Höhe des Heaps ist  $\lfloor \lg n \rfloor$ .
- Für  $\leq \lceil n/2^{h+1} \rceil$  Knoten mit Höhe  $h$  für  $h \geq 1$  benötigt Build-Max-Heap:

$$\sum_{h=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

## Bemerkungen:

- In der ersten Umstellung der Gleichung wird zunächst die Summe in das Bachmann-Landau-Symbol hineingezogen und gleichzeitig  $2n$  ausgeklammert. Die Konstante 2 kann daraufhin vernachlässigt werden.
- Die zweite Umstellung verallgemeinert die Schranken der Summe, um sie einer bekannten Reihe anzupassen.
- Für  $|x| < 1$  ist die geometrische Reihe definiert durch

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

Durch Ableiten und Multiplikation mit  $x$  beider Seiten der Gleichung als Äquivalenzumformungen erhalten wir

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

Aus  $x = 1/2$  folgt

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{1/2}{(1 - 1/2)^2} = 2.$$

# Einschub: Binary Heap

## Manipulation

### Algorithmen:

- **Maximum (Minimum)**

Gibt den Wert des Knotens mit dem größten (kleinsten) Sortierschlüssel zurück.

- **Extract-Max (Extract-Min)**

Extrahiert den Knoten mit dem größten (kleinsten) Schlüssel und gibt dessen Wert zurück.

- **Delete**

Löscht einen designierten Knoten.

- **Increase-Key (Decrease-Key)**

Erhöht (verringert) den Wert des Schlüssels eines Knotens.

- **Max-Insert (Min-Insert)**

Fügt einen neuen Knoten ein.

# Einschub: Binary Heap

## Manipulation

Algorithmus: Maximum

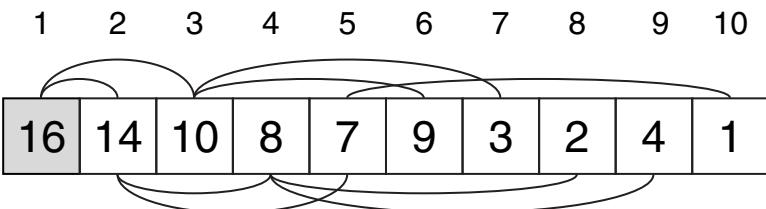
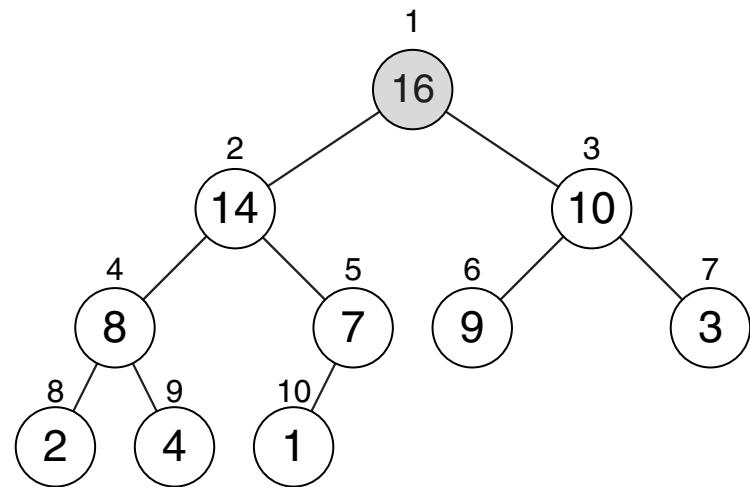
Eingabe: A. Heap (Array) mit  $n > 0$  Knoten.

Ausgabe: Wert des größten Knotens.

*Maximum(A)*

1. *return(A[1])*

**Beispiel:**



# Einschub: Binary Heap

## Manipulation

Algorithmus: Maximum

Eingabe: A. Heap (Array) mit  $n > 0$  Knoten.

Ausgabe: Wert des größten Knotens.

*Maximum(A)*

1. *return(A[1])*

Algorithmus: Extract-Max

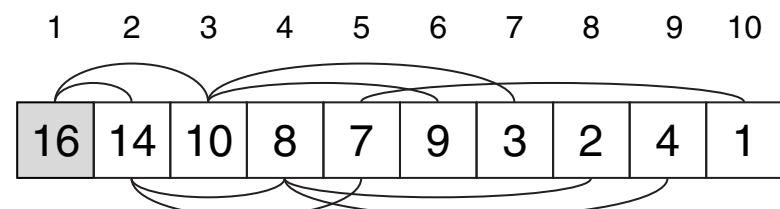
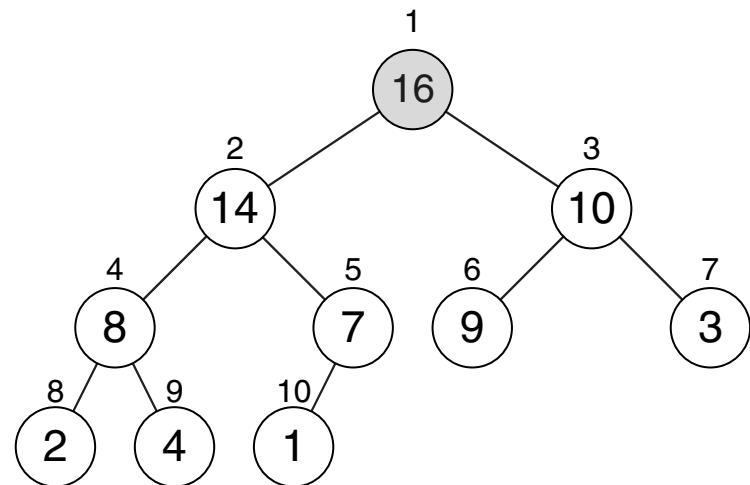
Eingabe: A. Heap (Array) mit  $n > 0$  Knoten.

Ausgabe: Wert des größten Knotens.

*ExtractMax(A)*

1. *max = A[1]*
2. *A[1] = A[n]*
3. *n = n - 1*
4. *MaxHeapify(A, 1)*
5. *return(max)*

**Beispiel:**



# Einschub: Binary Heap

## Manipulation

Algorithmus: Maximum

Eingabe: A. Heap (Array) mit  $n > 0$  Knoten.

Ausgabe: Wert des größten Knotens.

*Maximum(A)*

1. *return(A[1])*

Algorithmus: Extract-Max

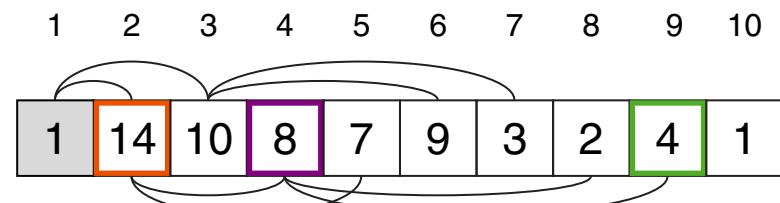
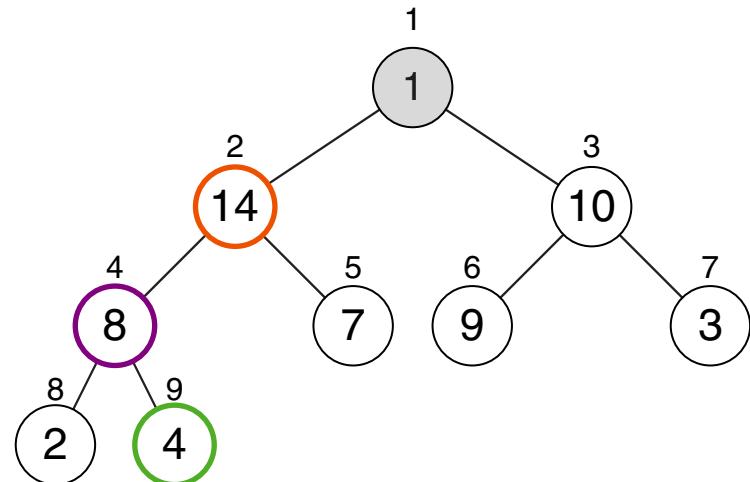
Eingabe: A. Heap (Array) mit  $n > 0$  Knoten.

Ausgabe: Wert des größten Knotens.

*ExtractMax(A)*

1. *max = A[1]*
2. *A[1] = A[n]*
3. *n = n - 1*
4. *MaxHeapify(A, 1)*
5. *return(max)*

**Beispiel:**



# Einschub: Binary Heap

## Manipulation

Algorithmus: Maximum

Eingabe: A. Heap (Array) mit  $n > 0$  Knoten.

Ausgabe: Wert des größten Knotens.

*Maximum(A)*

1. *return(A[1])*

Algorithmus: Extract-Max

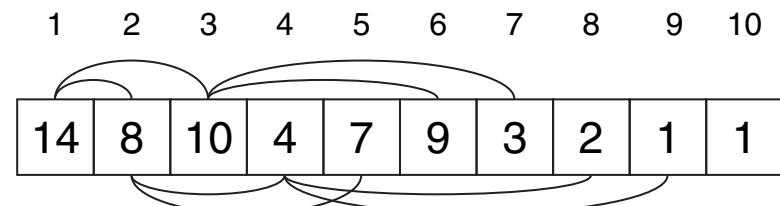
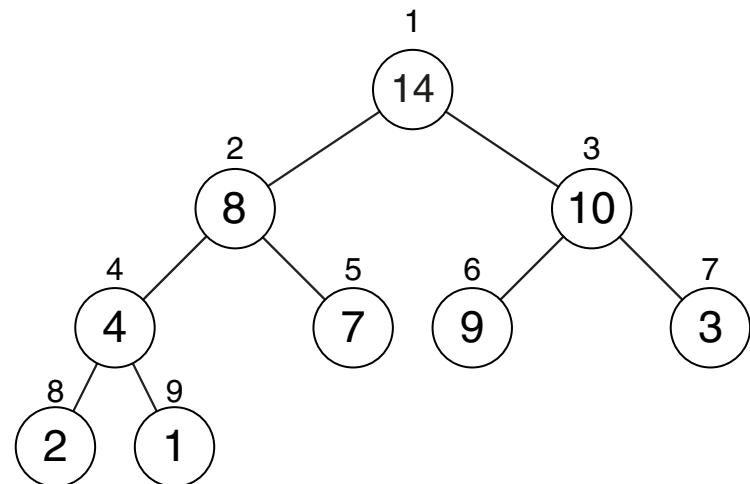
Eingabe: A. Heap (Array) mit  $n > 0$  Knoten.

Ausgabe: Wert des größten Knotens.

*ExtractMax(A)*

1. *max = A[1]*
2. *A[1] = A[n]*
3. *n = n - 1*
4. *MaxHeapify(A, 1)*
5. *return(max)*

**Beispiel:**



# Einschub: Binary Heap

## Manipulation

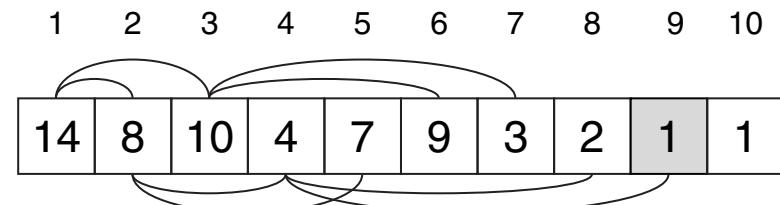
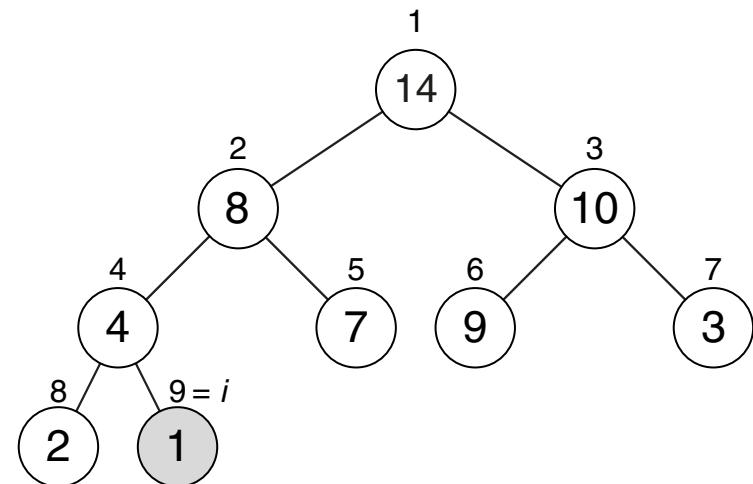
Algorithmus: Increase-Key

Eingabe: A. Heap (Array) mit  $n \geq i$  Knoten.  
i. Index des betroffenen Knotens.  
key. Neuer Schlüssel;  $A[i] \leq key$ .

*IncreaseKey*( $A, i, key$ )

1.  $A[i] = key$
2. **WHILE**  $i > 1$  **AND**  $A[\text{parent}(i)] < A[i]$  **DO**
3.   exchange  $A[i]$  with  $A[\text{parent}(i)]$
4.    $i = \text{parent}(i)$
5. **ENDDO**

**Beispiel:**



# Einschub: Binary Heap

## Manipulation

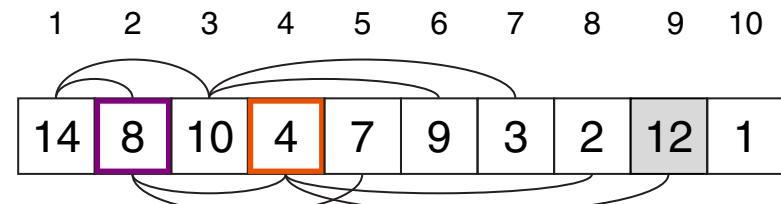
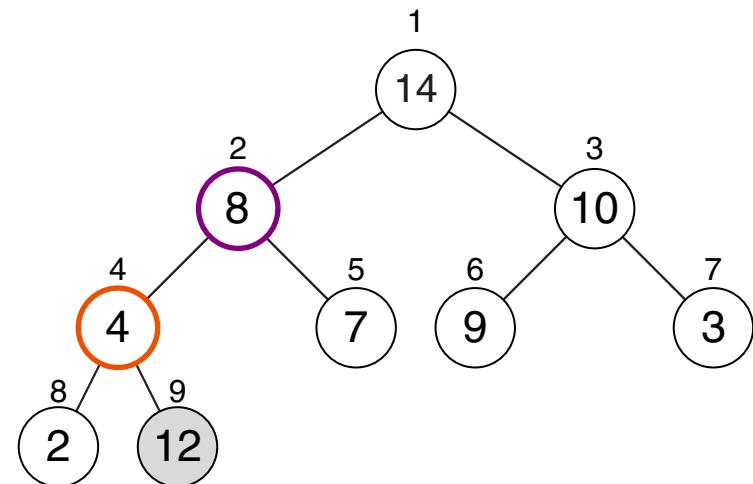
Algorithmus: Increase-Key

Eingabe: A. Heap (Array) mit  $n \geq i$  Knoten.  
i. Index des betroffenen Knotens.  
key. Neuer Schlüssel;  $A[i] \leq key$ .

*IncreaseKey*( $A, i, key$ )

1.  $A[i] = key$
2. **WHILE**  $i > 1$  **AND**  $A[\text{parent}(i)] < A[i]$  **DO**
3.   exchange  $A[i]$  with  $A[\text{parent}(i)]$
4.    $i = \text{parent}(i)$
5. **ENDDO**

**Beispiel:**



# Einschub: Binary Heap

## Manipulation

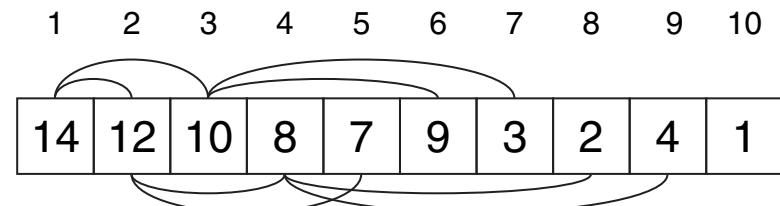
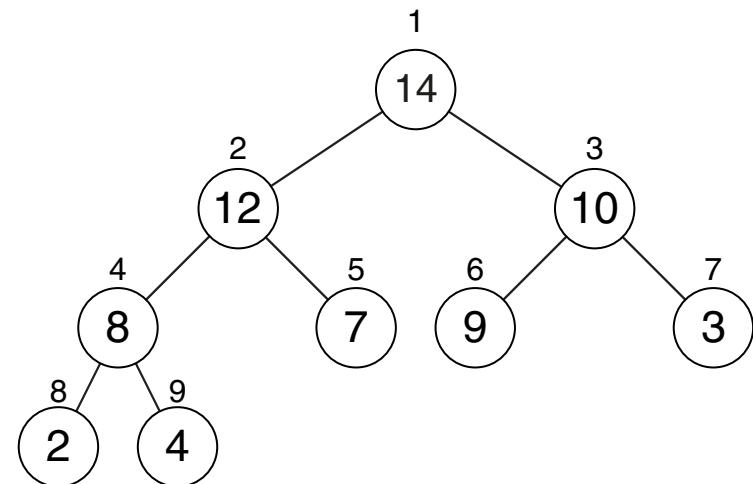
Algorithmus: Increase-Key

Eingabe: A. Heap (Array) mit  $n \geq i$  Knoten.  
i. Index des betroffenen Knotens.  
key. Neuer Schlüssel;  $A[i] \leq key$ .

*IncreaseKey*( $A, i, key$ )

1.  $A[i] = key$
2. **WHILE**  $i > 1$  **AND**  $A[\text{parent}(i)] < A[i]$  **DO**
3.   exchange  $A[i]$  with  $A[\text{parent}(i)]$
4.    $i = \text{parent}(i)$
5. **ENDDO**

**Beispiel:**



# Einschub: Binary Heap

## Manipulation

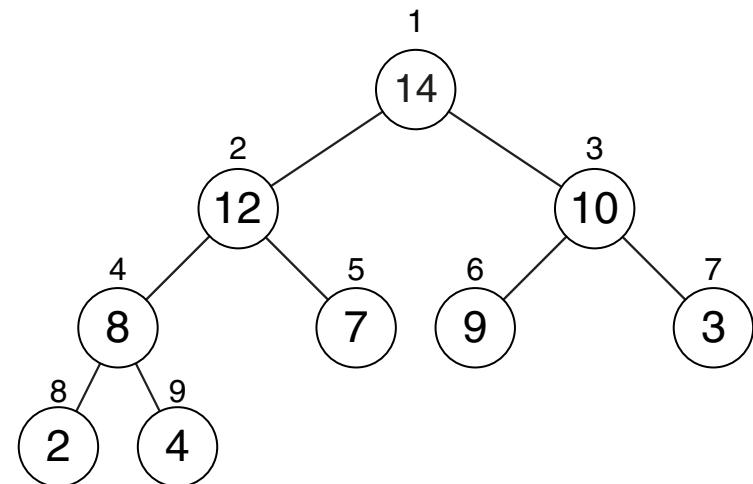
Algorithmus: Increase-Key

Eingabe: A. Heap (Array) mit  $n \geq i$  Knoten.  
i. Index des betroffenen Knotens.  
key. Neuer Schlüssel;  $A[i] \leq key$ .

*IncreaseKey*( $A, i, key$ )

1.  $A[i] = key$
2. **WHILE**  $i > 1$  **AND**  $A[\text{parent}(i)] < A[i]$  **DO**
3.   exchange  $A[i]$  with  $A[\text{parent}(i)]$
4.    $i = \text{parent}(i)$
5. **ENDDO**

**Beispiel:**

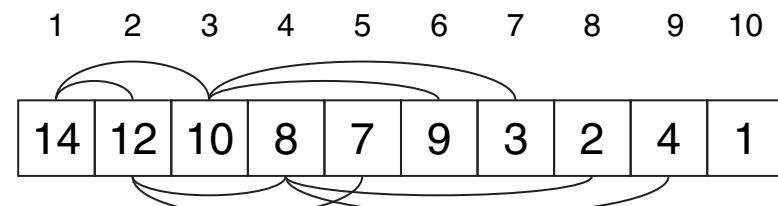


Algorithmus: Max-Insert

Eingabe: A. Heap mit  $n$  Knoten als Array  
der Länge  $m \geq n + 1$ .

*MaxInsert*( $A, key$ )

1.  $n = n + 1$
2.  $A[n] = -\infty$
3. *IncreaseKey*( $A, n, key$ )



## Bemerkungen:

- Laufzeiten der Manipulationsalgorithmen:
  - Maximum:  $\Theta(1)$
  - Extract-Max:  $O(\lg n)$  (Konstante Zuweisungen plus Laufzeit von Max-Heapify).
  - Increase-Key:  $O(\lg n)$  (Pfad von Knoten  $i$  zur Wurzel ist  $O(\lg n)$  lang für  $n$ -Knoten Heap).
  - Max-Insert:  $O(\lg n)$  (Konstante Zuweisungen plus Laufzeit von Increase-Key).

# Heapsort

## Algorithmus

Algorithmus: Heapsort

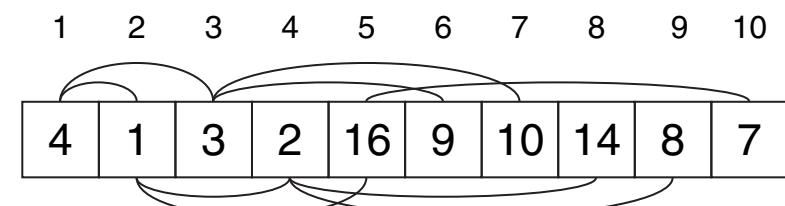
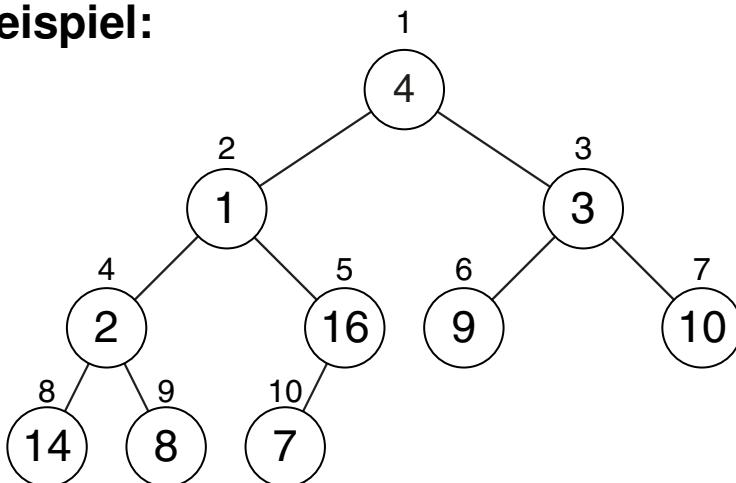
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.    $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

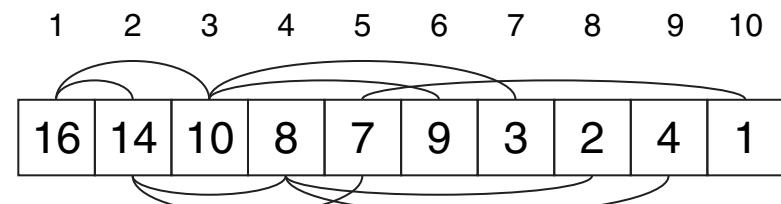
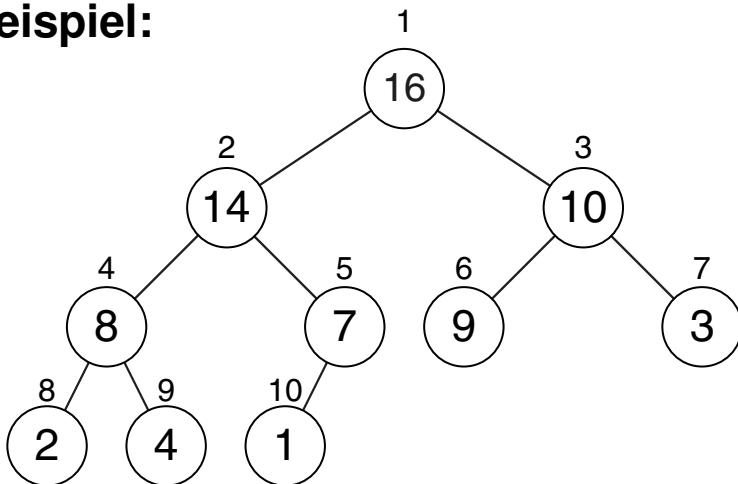
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTO** 2 **DO**
3.    $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

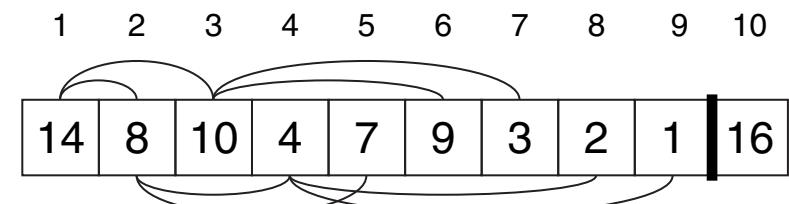
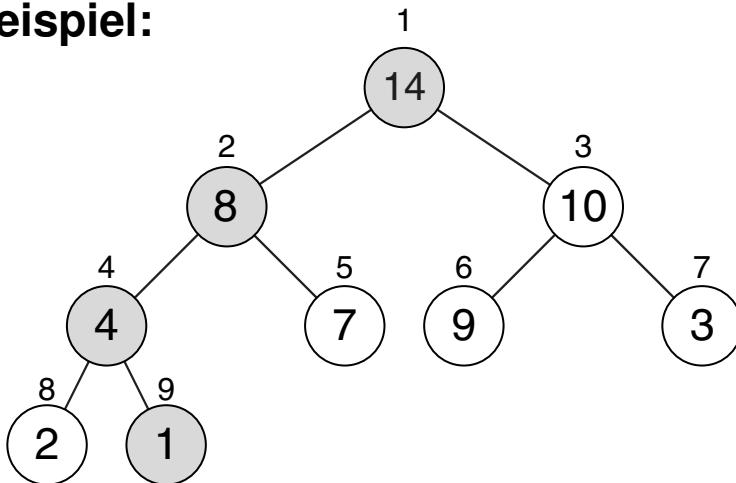
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTO** 2 **DO**
3.      $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

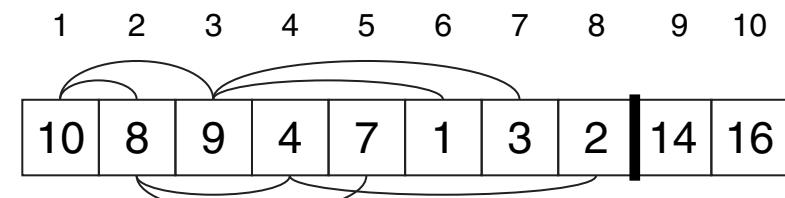
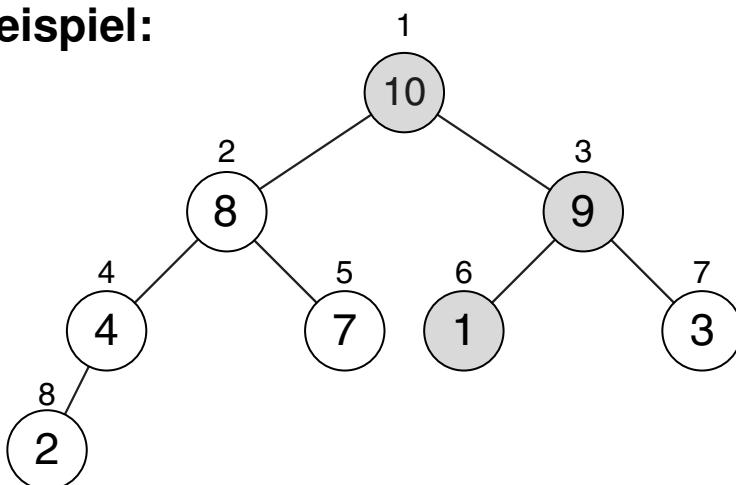
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTO** 2 **DO**
3.    $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

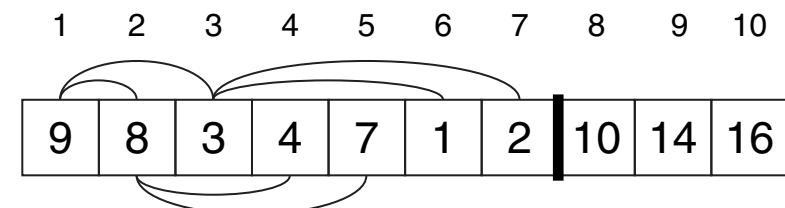
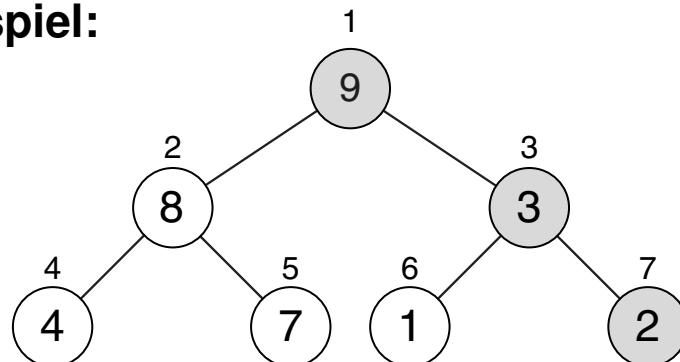
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.    $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

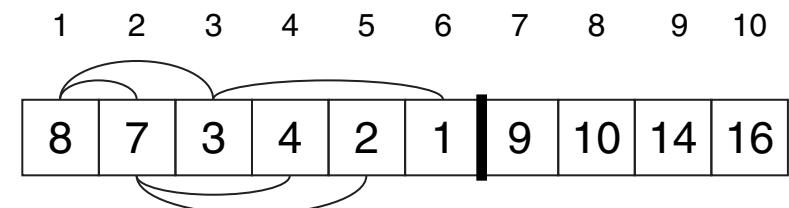
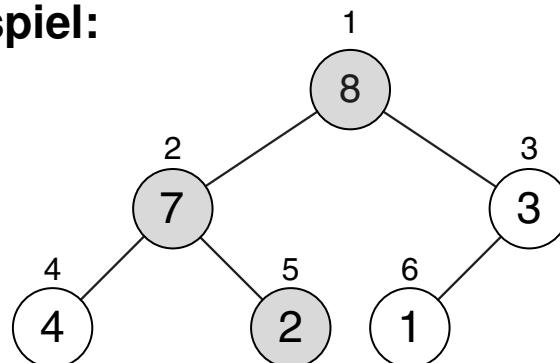
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.    $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

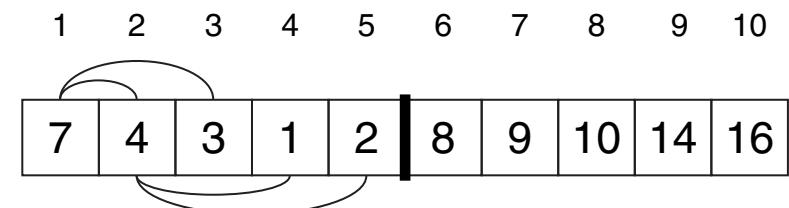
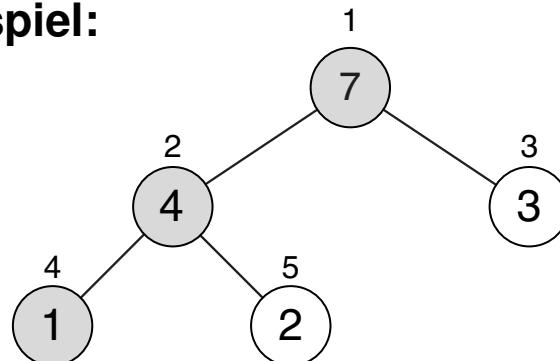
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.      $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

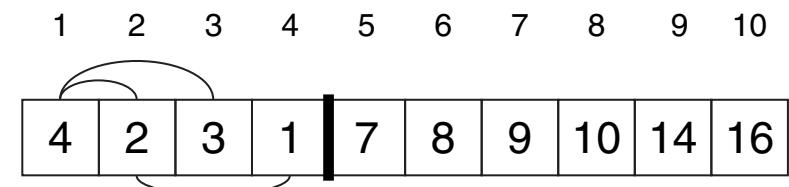
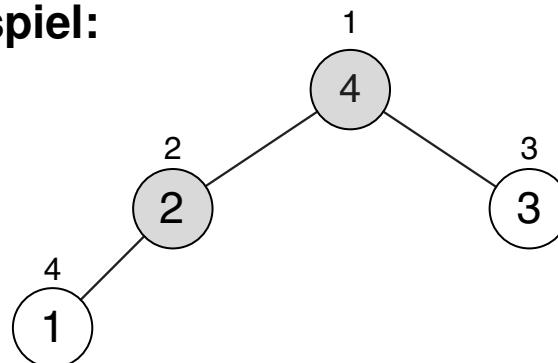
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTO** 2 **DO**
3.      $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

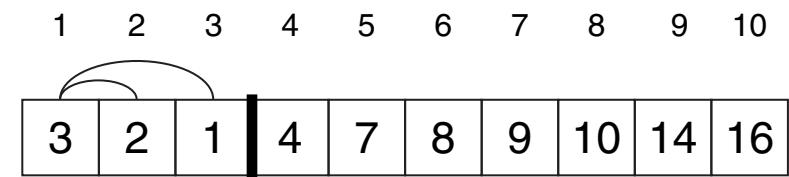
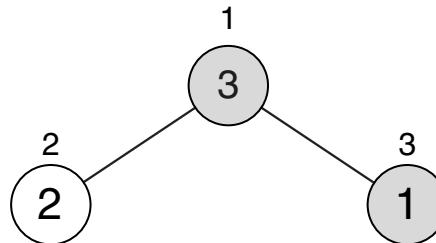
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.    $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

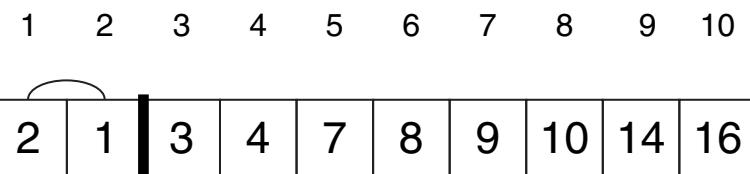
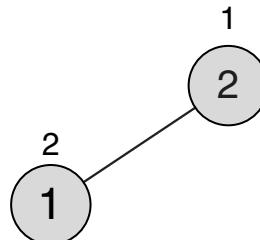
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.      $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



# Heapsort

## Algorithmus

Algorithmus: Heapsort

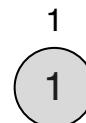
Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.      $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



1 2 3 4 5 6 7 8 9 10

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

# Heapsort

## Algorithmus

Algorithmus: Heapsort

Eingabe: A. Array der Länge  $n$ .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Heapsort(A)*

1. *BuildMaxHeap(A)*
2. **FOR**  $i = n$  **DOWNTTO** 2 **DO**
3.      $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Laufzeit:

- $O(n)$  Zeit für Build-MaxHeap
  - $n - 1$  mal  $O(\lg n)$  Zeit für Extract-Max (Zuweisung kostet  $\Theta(1)$ )
  - $n$  mal  $\Theta(1)$  für die For-Schleife.
- $T(n) = O(n \lg n)$

# Quicksort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortieren durch Vertauschen

# Quicksort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

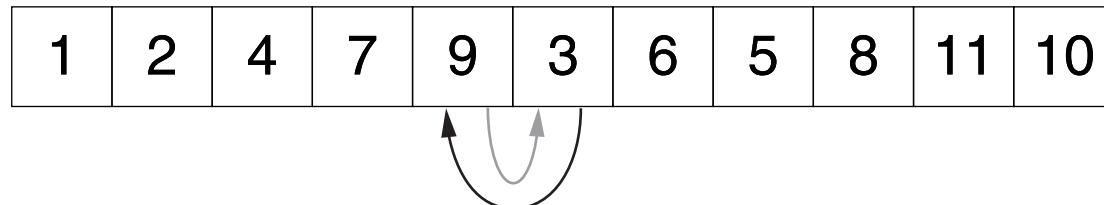
Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortieren durch Vertauschen

Naives Tauschverfahren: (wie bei Bubble Sort)

- Benachbarte Elemente solange wie nötig vergleichen und vertauschen.
- Quadratische Laufzeit.

Beispiel:



# Quicksort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Rekursives Divide and Conquer mit Vorsortieren durch Vertauschen

Rekursives Sortieren eines Arrays  $A$  der Länge  $n$ :

- $n = 1$ :  $A$  ist sortiert.
- $n > 1$ : Tausche Elemente in  $A$ , so dass  $A[1..q - 1] \leq A[q]$  und  $A[q + 1..n] \geq A[q]$  für ein  $0 < q \leq n$ ; dann sortiere die Teilarrays.

Voraussetzung:

- Die Vorsortierung des Arrays erzeugt im Schnitt gleich große Teilprobleme.

# Quicksort

## Algorithmus

Algorithmus: Quicksort.

Eingabe:

- A. Array von  $n$  Zahlen.
- p. Index ab dem A betrachtet wird.
- r. Index bis zu dem A betrachtet wird.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

*Quicksort*(A, p, r)

1. **IF**  $p < r$  **THEN**
2.      $q = \text{Partition}(A, p, r)$
3.     *Quicksort*(A, p,  $q - 1$ )
4.     *Quicksort*(A,  $q + 1$ , r)
5. **ENDIF**

# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

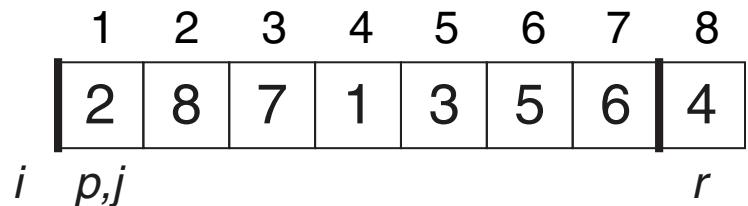
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

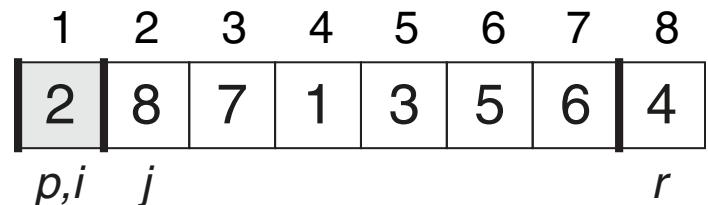
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

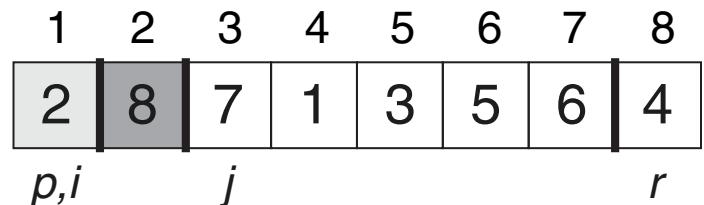
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

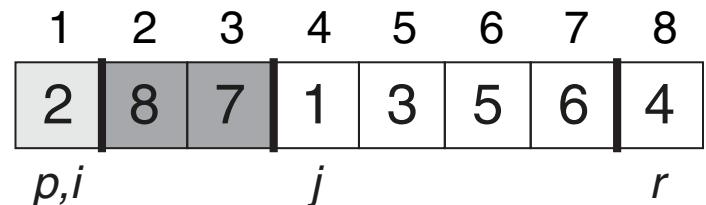
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

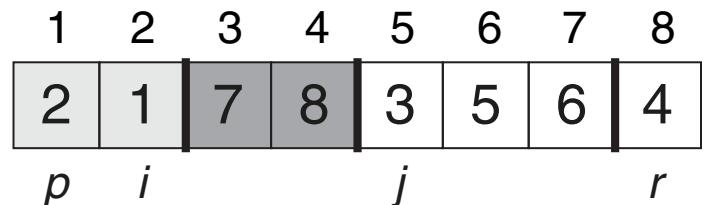
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

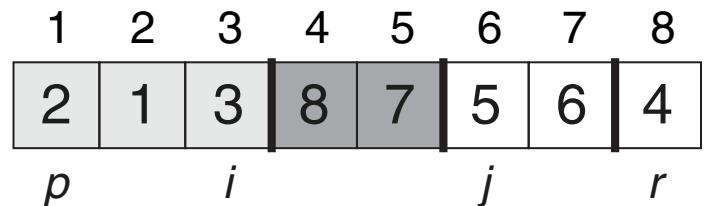
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

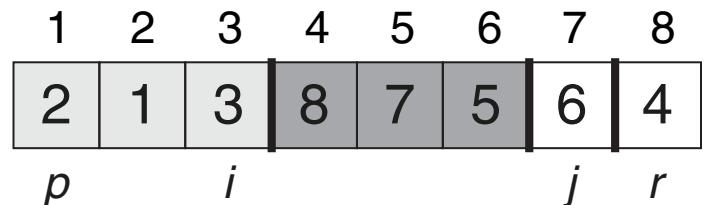
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

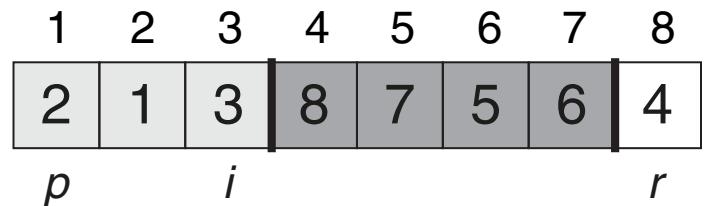
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

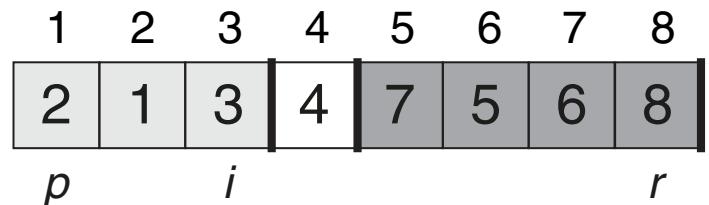
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Beispiel:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

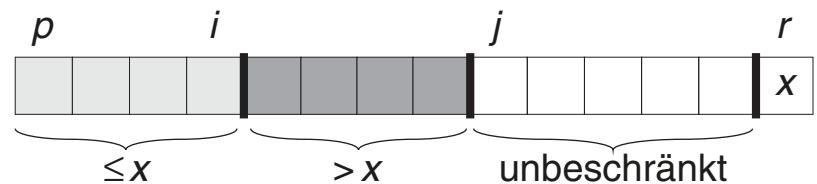
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Schematisch:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

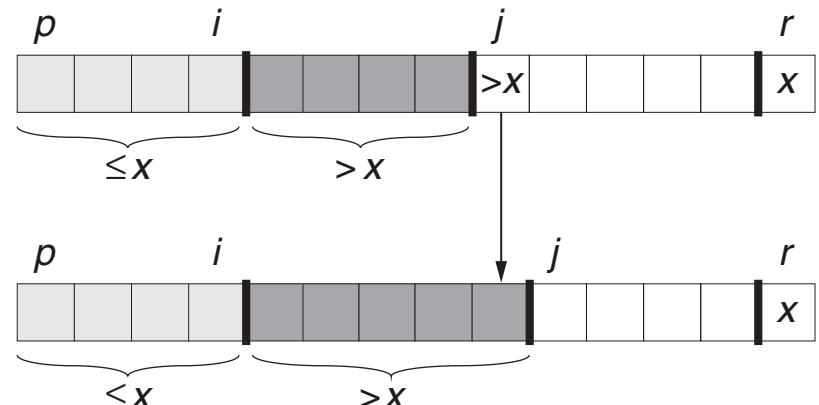
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Schematisch:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

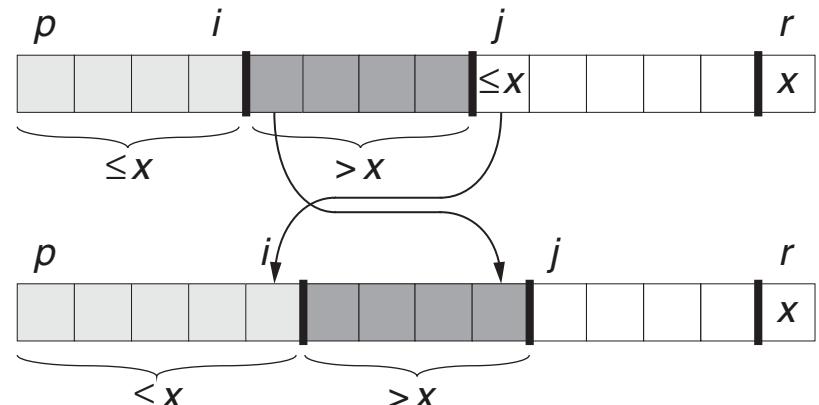
$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Schematisch:**



# Quicksort

## Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von  $m$  Zahlen.

$p, r$ . Indexe, so dass  $0 < p \leq r \leq m$ .

Ausgabe: Index  $q$ , so dass  $p \leq q \leq r$  und  $A[i] \leq A[q]$  für alle  $i \in [p, q - 1]$  und  $A[q] \leq A[j]$  für alle  $j \in [q + 1, r]$ .

*Partition*( $A, p, r$ )

1.  $x = A[r]$
2.  $i = p - 1$
3. **FOR**  $j = p$  **TO**  $r - 1$  **DO**
4.     **IF**  $A[j] \leq x$  **THEN**
5.          $i = i + 1$
6.         exchange  $A[i]$  with  $A[j]$
7.     **ENDIF**
8. **ENDDO**
9. exchange  $A[i + 1]$  with  $A[r]$
10. **return**( $i + 1$ )

**Laufzeit:**

- Eingabegröße:  $n = r - p + 1$ .
  - Vertauschen von Elementen:  $\Theta(1)$ .
  - Die For-Schleife wird  $n$  mal ausgeführt.
  - Alle übrigen Anweisungen:  $\Theta(1)$ .
  - Der Rumpf der For-Schleife ist in  $\Theta(1)$ .
- *Partition* ist in  $\Theta(n)$ .

## Bemerkungen:

- Das Vorsortieren von  $A$  wird als „Partitionieren“ bezeichnet: Das Array wird in zwei Partitionen unterteilt.
- Das Element aus  $A$ , dessen Wert die beiden Partitionen trennt, heißt Pivot-Element.
- Zur Laufzeit teilt *Partition* das Array  $A$  in vier Bereiche: Alle Elemente in  $A[p..i]$  sind kleiner oder gleich  $x$ , alle Elemente in  $A[i + 1..j - 1]$  sind größer als  $x$ , die Elemente in  $A[j..r - 1]$  können jeden beliebigen Wert haben, und  $A[r] = x$  ist das Pivot-Element.
- In jeder Iteration der For-Schleife wird das aktuell betrachtete Element  $j$  einem der beiden Partitionen hinzugefügt. Wenn  $A[j] > x$ , dann genügt das Inkrementieren von  $j$ , andernfalls wird das erste Element der zweiten Partition  $A[i + 1]$  mit  $A[j]$  vertauscht, und  $i$  inkrementiert. Da keine der beiden Partitionen sortiert sind, müssen die Elemente der zweiten Partition nicht nach rechts verschoben werden.
- Zuletzt wird das Pivot-Element mittig zwischen die beiden Partitionen getauscht, um die Vorsortierung abzuschließen.
- Dieser Partitionierungsalgorithmus wurde von Nico Lomuto vorgeschlagen.
- Charle Antony Richard Hoare, Erfinder von Quicksort, hat ein anderes Verfahren vorgeschlagen, bei dem das Pivot-Element mittig im Array liegt.

# Quicksort

## Laufzeit (informell)

Die Laufzeit von Quicksort hängt vom Größenverhältnis der Partitionen ab.

Worst-Case-Partitionierung:

- Immer maximal unbalancierte Partitionen: 0 Elemente vs.  $n - 1$  Elemente.
- $$\begin{aligned} T(n) &= T(n - 1) + T(0) + \Theta(n) \\ &= T(n - 1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$
 (Substitutionsmethode mit Hypothese  $dn^2 - d'n$ )
- Tritt ein, wenn die Probleminstanz sortiert ist.

Best-Case-Partitionierung:

- Immer balancierte Partitionen:  $\lfloor n/2 \rfloor$  Elemente vs.  $\lceil n/2 \rceil - 1$  Elemente.
- $$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \quad (\text{Vereinfachung}) \\ &= \Theta(n \lg n) \end{aligned}$$
 (Fall 2 des Master-Theorems)
- Tritt ein, wenn der Wert des Pivot-Elements der Median aller Werte in  $A$  ist.

# Quicksort

## Laufzeit (informell)

Statische Partitionierung:

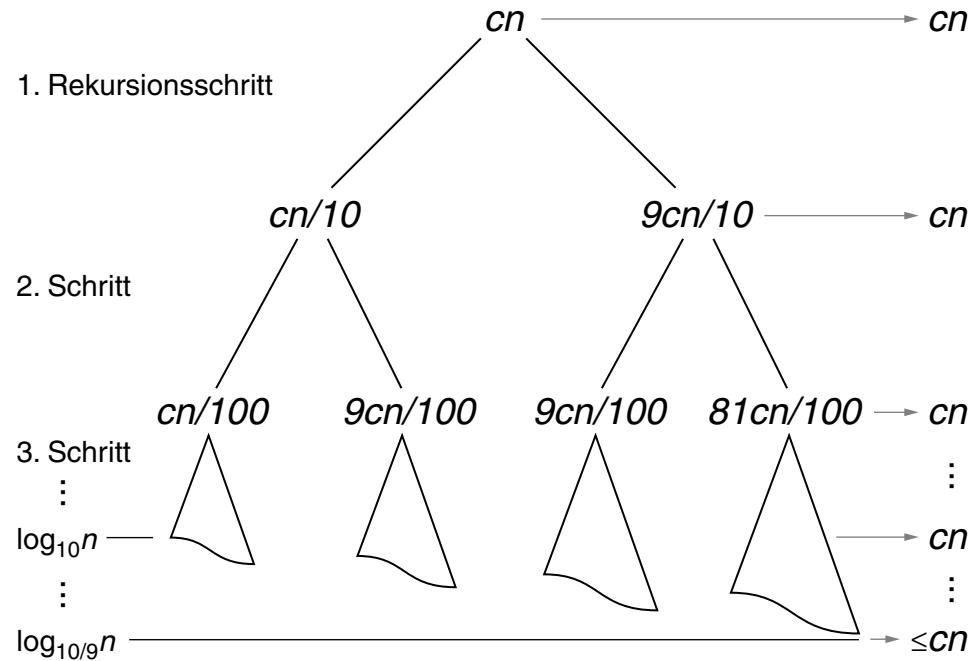
- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $T(n) = T(n/10) + T(9n/10) + \Theta(n)$

# Quicksort

## Laufzeit (informell)

Statische Partitionierung:

- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $T(n) = T(n/10) + T(9n/10) + \Theta(n)$



# Quicksort

## Laufzeit (informell)

Statische Partitionierung:

- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= O(n \lg n) \end{aligned}$$
 (Abschätzung nach Rekursionsbaummethode)
- Jede Partitionierung mit konstantem Verhältnis führt zur Laufzeit  $O(n \lg n)$ .

# Quicksort

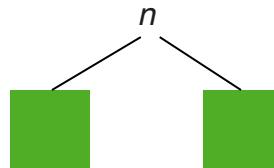
## Laufzeit (informell)

Statische Partitionierung:

- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= O(n \lg n) \end{aligned}$$
 (Abschätzung nach Rekursionsbaummethode)
- Jede Partitionierung mit konstantem Verhältnis führt zur Laufzeit  $O(n \lg n)$ .

Average-Case-Partitionierung: (intuitiv)

- Es ist ein Mix aus „guten“ und „schlechten“ Partitionierungen zu erwarten.



# Quicksort

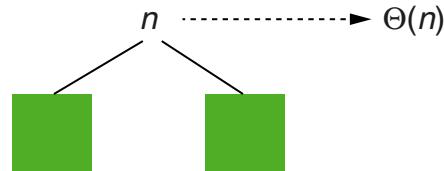
## Laufzeit (informell)

Statische Partitionierung:

- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= O(n \lg n) \end{aligned}$$
 (Abschätzung nach Rekursionsbaummethode)
- Jede Partitionierung mit konstantem Verhältnis führt zur Laufzeit  $O(n \lg n)$ .

Average-Case-Partitionierung: (intuitiv)

- Es ist ein Mix aus „guten“ und „schlechten“ Partitionierungen zu erwarten.



# Quicksort

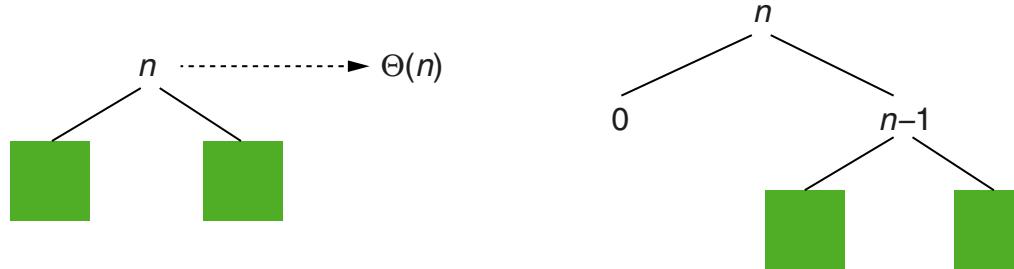
## Laufzeit (informell)

Statische Partitionierung:

- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= O(n \lg n) \end{aligned}$$
 (Abschätzung nach Rekursionsbaummethode)
- Jede Partitionierung mit konstantem Verhältnis führt zur Laufzeit  $O(n \lg n)$ .

Average-Case-Partitionierung: (intuitiv)

- Es ist ein Mix aus „guten“ und „schlechten“ Partitionierungen zu erwarten.



# Quicksort

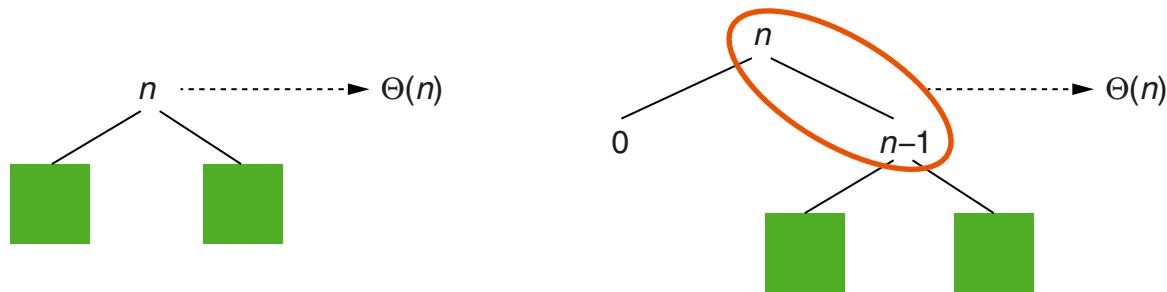
## Laufzeit (informell)

Statische Partitionierung:

- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= O(n \lg n) \end{aligned}$$
 (Abschätzung nach Rekursionsbaummethode)
- Jede Partitionierung mit konstantem Verhältnis führt zur Laufzeit  $O(n \lg n)$ .

Average-Case-Partitionierung: (intuitiv)

- Es ist ein Mix aus „guten“ und „schlechten“ Partitionierungen zu erwarten.



- Beide Situationen sind in  $O(n \lg n)$ ; der Unterschied ist ein konstanter Faktor.

# Quicksort

## Pivot-Selektion

Die Größe der Partitionen hängt vom gewählten Pivot-Element ab:

- **Statisches Element**

Ein vordefiniertes Element wird als Pivot-Element verwendet, zum Beispiel das erste, mittige oder letzte.

- **Zufallselement**

Das Element, das durch eine Zufallszahl zwischen  $p$  und  $r$  bestimmt wird.

Alternative: Das Array zufällig permutieren und ein statisches Element verwenden.

- **Median-of- $n$**

Ziehe eine Stichprobe von  $n$  Elementen des Arrays, sortiere sie und bestimme den Median als Pivot-Element. Nach Sedgewick liefert  $n = 3$  im Schnitt den größten Gewinn. Für große Arrays werden größere Werte von  $n$  gewählt.

Abhängig vom eingesetzten Partitionierungsalgorithmus, muss das gewählte Pivot-Element zunächst an die richtige Position getauscht werden.

Mit einem zufällig gewählten Pivot-Element ist die Average-Case-Laufzeit von Quicksort beweisbar in  $O(n \lg n)$ .

# Minimales vergleichsbasiertes Sortieren

## Entscheidungsbaummodell

Was ist die kleinstmögliche Maximalzahl an Vergleichen, um  $n$  Zahlen zu sortieren?

Beispiel  $n = 3$ :

1:2

	1	2	3
A	$a_1$	$a_2$	$a_3$

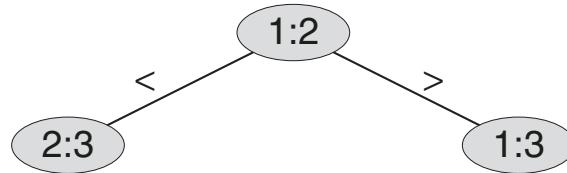
- Darstellung des Vorgehens eines hypothetischen Algorithmus.
- $i : j$  steht für den Vergleich von  $A[i] = a_i$  mit  $A[j] = a_j$ .

# Minimales vergleichsbasiertes Sortieren

## Entscheidungsbaummodell

Was ist die kleinstmögliche Maximalzahl an Vergleichen, um  $n$  Zahlen zu sortieren?

Beispiel  $n = 3$ :



	1	2	3
$A$	$a_1$	$a_2$	$a_3$

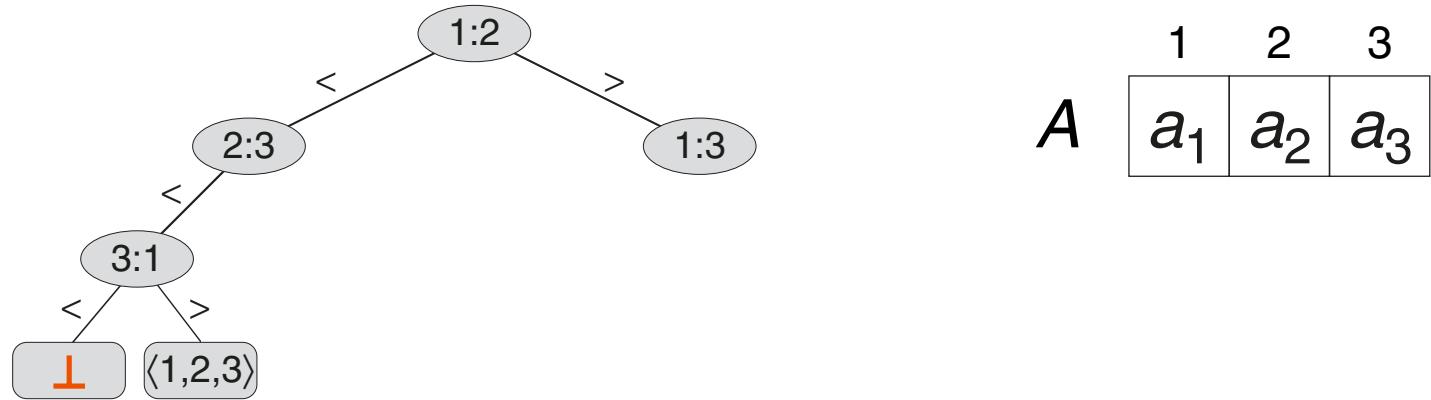
- Darstellung des Vorgehens eines hypothetischen Algorithmus.
- $i : j$  steht für den Vergleich von  $A[i] = a_i$  mit  $A[j] = a_j$ .

# Minimales vergleichsbasiertes Sortieren

## Entscheidungsbaummodell

Was ist die kleinstmögliche Maximalzahl an Vergleichen, um  $n$  Zahlen zu sortieren?

Beispiel  $n = 3$ :



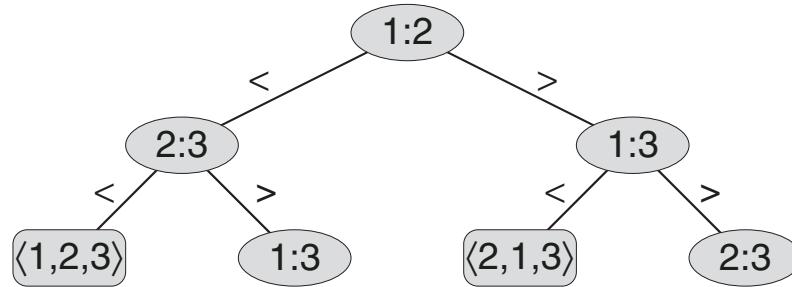
- Darstellung des Vorgehens eines hypothetischen Algorithmus.
- $i : j$  steht für den Vergleich von  $A[i] = a_i$  mit  $A[j] = a_j$ .
- **Unmögliche Situation:**  $a_1 < a_2 < a_3 < a_1$ .
- Redundanter Vergleich (Transitivität): Aus  $a_1 < a_2$  und  $a_2 < a_3$  folgt  $a_1 < a_3$ .
- Blattknoten  $\langle i, j, k \rangle$  stehen für Permutationen der Probleminstanz  $[a_i, a_j, a_k]$ .

# Minimales vergleichsbasiertes Sortieren

## Entscheidungsbaummodell

Was ist die kleinstmögliche Maximalzahl an Vergleichen, um  $n$  Zahlen zu sortieren?

Beispiel  $n = 3$ :



A	1	2	3
	$a_1$	$a_2$	$a_3$

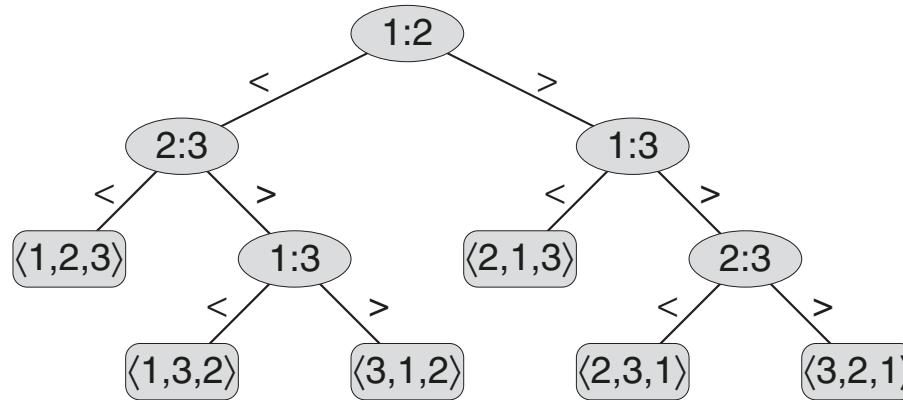
- Darstellung des Vorgehens eines hypothetischen Algorithmus.
- $i : j$  steht für den Vergleich von  $A[i] = a_i$  mit  $A[j] = a_j$ .
- Blattknoten  $\langle i, j, k \rangle$  stehen für Permutationen der Probleminstanz  $[a_i, a_j, a_k]$ .

# Minimales vergleichsbasiertes Sortieren

## Entscheidungsbaummodell

Was ist die kleinstmögliche Maximalzahl an Vergleichen, um  $n$  Zahlen zu sortieren?

Beispiel  $n = 3$ :



	1	2	3
$A$	$a_1$	$a_2$	$a_3$

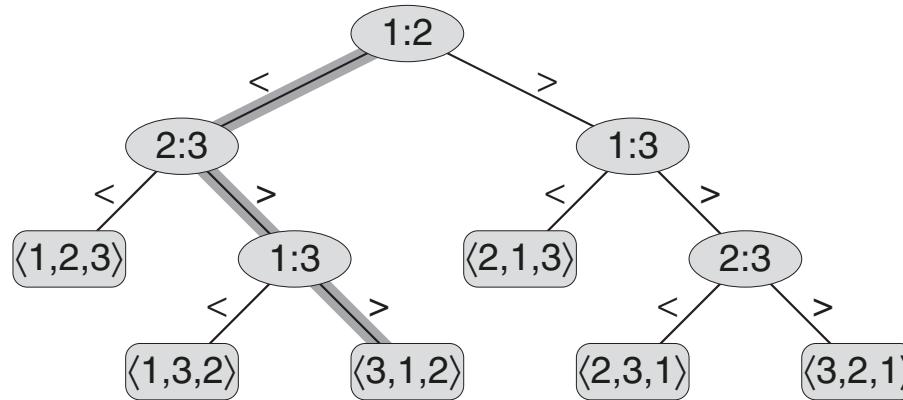
- Darstellung des Vorgehens eines hypothetischen Algorithmus.
- $i : j$  steht für den Vergleich von  $A[i] = a_i$  mit  $A[j] = a_j$ .
- Blattknoten  $\langle i, j, k \rangle$  stehen für Permutationen der Probleminstanz  $[a_i, a_j, a_k]$ .
- Es gibt mindestens  $n!$  Blattknoten; mind. einen für jede mögliche Permutation.

# Minimales vergleichsbasiertes Sortieren

## Entscheidungsbaummodell

Was ist die kleinstmögliche Maximalzahl an Vergleichen, um  $n$  Zahlen zu sortieren?

Beispiel  $n = 3$ :



	1	2	3
A	6	8	5

- Darstellung des Vorgehens eines hypothetischen Algorithmus.
- $i : j$  steht für den Vergleich von  $A[i] = a_i$  mit  $A[j] = a_j$ .
- Blattknoten  $\langle i, j, k \rangle$  stehen für Permutationen der Probleminstanz  $[a_i, a_j, a_k]$ .
- Es gibt mindestens  $n!$  Blattknoten; mind. einen für jede mögliche Permutation.
- Sortieren bedeutet, den Pfad zur richtigen Permutation zu ermitteln.

# Minimales vergleichsbasiertes Sortieren

## Satz 1 (Untere Schranke für den Worst Case für vergleichsbasiertes Sortieren)

Jeder vergleichsbasierte Sortieralgorithmus benötigt im Worst-Case  $\Omega(n \lg n)$  Vergleiche.

# Minimales vergleichsbasiertes Sortieren

## Satz 1 (Untere Schranke für den Worst Case für vergleichsbasiertes Sortieren)

Jeder vergleichsbasierte Sortieralgorithmus benötigt im Worst-Case  $\Omega(n \lg n)$  Vergleiche.

### Beweis:

Für das Sortieren von  $n$  Elementen gibt es einen Entscheidungsbaum der Höhe  $h$  mit  $l$  Blattknoten. Da alle  $n!$  möglichen Permutationen vorkommen müssen, gilt:

$$n! \leq l \leq 2^h$$

# Minimales vergleichsbasiertes Sortieren

## Satz 1 (Untere Schranke für den Worst Case für vergleichsbasiertes Sortieren)

Jeder vergleichsbasierte Sortieralgorithmus benötigt im Worst-Case  $\Omega(n \lg n)$  Vergleiche.

### Beweis:

Für das Sortieren von  $n$  Elementen gibt es einen Entscheidungsbaum der Höhe  $h$  mit  $l$  Blattknoten. Da alle  $n!$  möglichen Permutationen vorkommen müssen, gilt:

$$\begin{aligned} n! &\leq l \leq 2^h \\ \Leftrightarrow 2^h &\geq n! \\ \Leftrightarrow h &\geq \lg(n!) \end{aligned}$$

# Minimales vergleichsbasiertes Sortieren

## Satz 1 (Untere Schranke für den Worst Case für vergleichsbasiertes Sortieren)

Jeder vergleichsbasierte Sortieralgorithmus benötigt im Worst-Case  $\Omega(n \lg n)$  Vergleiche.

### Beweis:

Für das Sortieren von  $n$  Elementen gibt es einen Entscheidungsbaum der Höhe  $h$  mit  $l$  Blattknoten. Da alle  $n!$  möglichen Permutationen vorkommen müssen, gilt:

$$\begin{aligned} n! &\leq l \leq 2^h \\ \Leftrightarrow 2^h &\geq n! \\ \Leftrightarrow h &\geq \lg(n!) \\ &= \lg((n/e)^n) && \text{Stirling-Formel} \\ &= n \lg((n/e)) \\ &= n \lg n - n \lg e \end{aligned}$$

# Minimales vergleichsbasiertes Sortieren

## Satz 1 (Untere Schranke für den Worst Case für vergleichsbasiertes Sortieren)

Jeder vergleichsbasierte Sortieralgorithmus benötigt im Worst-Case  $\Omega(n \lg n)$  Vergleiche.

### Beweis:

Für das Sortieren von  $n$  Elementen gibt es einen Entscheidungsbaum der Höhe  $h$  mit  $l$  Blattknoten. Da alle  $n!$  möglichen Permutationen vorkommen müssen, gilt:

$$\begin{aligned} n! &\leq l \leq 2^h \\ \Leftrightarrow 2^h &\geq n! \\ \Leftrightarrow h &\geq \lg(n!) \\ &= \lg((n/e)^n) && \text{Stirling-Formel} \\ &= n \lg((n/e)) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \quad \square \end{aligned}$$

## Bemerkungen:

- Es werden nur Algorithmen betrachtet, die vergleichsbasiert sortieren.
- Es werden nur Probleminstanzen betrachtet, in denen  $n$  verschiedene Elemente zu sortieren sind. Das Modell ist verallgemeinerbar auf Probleminstanzen mit mehrfach vorkommenden Elementen.
- Es werden nur die Vergleichsoperationen  $<$  und  $>$  betrachtet. Alle übrigen Anweisungen, zum Beispiel für das Austauschen von Array-Elementen oder zum Kontrollfluss, werden ignoriert.
- Die Zahlen in Knoten beziehen sich auf die Array-Indexe der initialen Probleminstanz. Zwischenzeitliches Verschieben von Array-Elementen wird nicht betrachtet.
- Jeder direkte Pfad von der Wurzel zu einem Blattknoten im Entscheidungsbaum beschreibt eine Folge von Vergleichen, die ein Algorithmus durchführt, um zu der Entscheidung zu gelangen, dass die durch den Blattknoten beschriebene Permutation das Problem löst.

# Counting Sort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortieren durch Abzählen von Elementen.

# Counting Sort

## Algorithmus

Problem: Sortieren

Instanz: A. Folge von  $n$  Zahlen  $A = (a_1, a_2, \dots, a_n)$ .

Lösung: Eine Permutation  $A' = (a'_1, a'_2, \dots, a'_n)$  von  $A$ , so dass  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz  $A$  löst.

Idee: Sortieren durch Abzählen von Elementen.

Voraussetzung:

- Alle zu sortierenden Elemente sind im Intervall  $[0, k]$ .
- Die Position des  $i$ -ten Elements von  $A$  ist damit durch Abzählen bestimmbar.

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in  $A$ .

Ausgabe: Eine aufsteigend sortierte Permutation von  $A$  in  $B$ .

*CountingSort( $A, B, k$ )*

1.  $C = \text{array}(k)$
2. **FOR**  $i = 1$  **TO**  $n$  **DO**
3.      $C[A[i]] = C[A[i]] + 1$
4. **ENDDO**
5. **FOR**  $i = 1$  **TO**  $k$  **DO**
6.      $C[i] = C[i] + C[i - 1]$
7. **ENDDO**
8. **FOR**  $i = n$  **DOWNTTO** 1 **DO**
9.      $B[C[A[i]]] = A[i]$
10.     $C[A[i]] = C[A[i]] - 1$
11. **ENDDO**

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- c. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8
B								
	0	1	2	3	4	5		
C								

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8
B								
	0	1	2	3	4	5		
C	2	0	2	3	0	1		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8
B								
	0	1	2	3	4	5		
C	2	2	4	7	7	8		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- c. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

									i
A	2	5	3	0	2	3	0	3	8
	1	2	3	4	5	6	7	8	
B							7		
	1	2	3	4	5	6			
C	2	2	4	7	7	8			
	0	1	2	3	4	5			

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	$i$							
$A$	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
$B$	1	2	3	4	5	6	7	8
								3
$C$	0	1	2	3	4	5		
	2	2	4	6	7	8		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	$i$							
A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8
B		0						3
	0	1	2	3	4	5	6	7
C	1	2	4	6	7	8		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	$i$							
$A$	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
$B$	1	2	3	4	5	6	7	8
	0					3	3	
$C$	0	1	2	3	4	5		
	1	2	4	5	7	8		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	$i$							
$A$	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
$B$	1	2	3	4	5	6	7	8
	0		2		3	3		
$C$	0	1	2	3	4	5		
	1	2	3	5	7	8		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	$i$							
$A$	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
$B$	1	2	3	4	5	6	7	8
	0	0		2		3	3	
$C$	0	1	2	3	4	5		
	0	2	3	5	7	8		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	$i$							
$A$	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
$B$	1	2	3	4	5	6	7	8
	0	0		2	3	3	3	
$C$	0	1	2	3	4	5		
	0	2	3	4	7	8		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

$i$	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8
B	0	0		2	3	3	3	5
	0	1	2	3	4	5		
C	0	2	3	4	7	7		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- c. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

*CountingSort(A, B, k)*

```
1. C = array(k)
2. FOR i = 1 TO n DO
3.   C[A[i]] = C[A[i]] + 1
4. ENDDO
5. FOR i = 1 TO k DO
6.   C[i] = C[i] + C[i - 1]
7. ENDDO
8. FOR i = n DOWNTTO 1 DO
9.   B[C[A[i]]] = A[i]
10.  C[A[i]] = C[A[i]] - 1
11. ENDDO
```

**Beispiel:**

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5
	0	1	2	3	4	5		
C	0	2	2	4	7	7		

# Counting Sort

## Algorithmus

Algorithmus: Counting Sort.

Eingabe:

- A. Array von  $n$  ganzen Zahlen.
- B. Array der Länge  $n$  als Ausgabe.
- k. Wert des größten Elements in  $A$ .

Ausgabe: Eine aufsteigend sortierte Permutation von  $A$  in  $B$ .

*CountingSort*( $A, B, k$ )

```
1.  $C = \text{array}(k)$ 
2. FOR  $i = 1$  TO  $n$  DO
3.    $C[A[i]] = C[A[i]] + 1$ 
4. ENDDO
5. FOR  $i = 1$  TO  $k$  DO
6.    $C[i] = C[i] + C[i - 1]$ 
7. ENDDO
8. FOR  $i = n$  DOWNTTO 1 DO
9.    $B[C[A[i]]] = A[i]$ 
10.   $C[A[i]] = C[A[i]] - 1$ 
11. ENDDO
```

**Laufzeit:**

- Zwei For-Schleifen in  $\Theta(n)$ , eine in  $\Theta(k)$ .  
→  $T(n) = \Theta(n + k)$
- Wenn  $k = O(n)$ , dann  $T(n) = \Theta(n)$ .

**Platz:**

- $S(n) = \Theta(n + k)$

**Eigenschaften:**

- **Stabilität:** Die Reihenfolge gleicher Elemente in  $A$  bleibt erhalten.