

Kapitel ADS:III

III. Sortieren

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

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**

Vergleiche mit Quicksort.

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**

Beispiel:

(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**

Beispiel:

	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**

Beispiel:

	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**

Beispiel:

	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, 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**

Beispiel:

	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, 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**

Beispiel:

	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, 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**

Beispiel:

	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**

Beispiel:

	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								
$(A, 3, 3)$	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**

Beispiel:

(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**

Beispiel:

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)										
2	4	7	1	4	6					
(A, 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**

Beispiel:

(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**

Beispiel:

(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**

Beispiel:

(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			

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**

Laufzeit:

- Zeilen 1, 2 und 6: $\Theta(1)$
 - Merge: $\Theta(n)$
- $T(n) = 2T(n/2) + \Theta(n)$
- $T(n) = \Theta(n \lg n)$
(gemäß Fall 2 des Master-Theorems)

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**
- | ...
23. **ENDDO**

Eigenschaften:

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

Beispiel:

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

	8	9	10	11	12	13	14	15	16	17
A	...	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 ∞ am Ende von L und R vermeidet wiederholtes Prüfen auf Überschreitung von n_1 und n_2 durch i und j .

Beispiel:

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

	8	9	10	11	12	13	14	15	16	17
A	...	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 ∞ am Ende von L und R vermeidet wiederholtes Prüfen auf Überschreitung von n_1 und n_2 durch i und j .

Beispiel:

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

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

L	1	2	3	4	5	
	2	4	5	7		

R	1	2	3	4	5

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 ∞ am Ende von L und R vermeidet wiederholtes Prüfen auf Überschreitung von n_1 und n_2 durch i und j .

Beispiel:

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

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

L	1	2	3	4	5	
	2	4	5	7		

R	1	2	3	4	5	
	1	2	3	6		

Merge Sort

Merge

Merge(A, p, q, r)

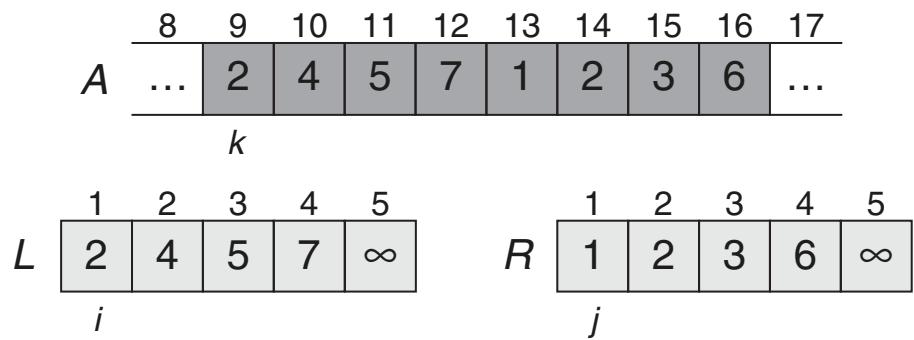
1. $n_1 = q - p + 1$
2. $n_2 = r - p$
3. $L = \text{array}(n_1 + 1)$
4. $R = \text{array}(n_2 + 1)$
5. **FOR** $i = 1$ **TO** n_1 **DO**
6. $L[i] = A[p + i - 1]$
7. **ENDDO**
8. **FOR** $j = 1$ **TO** n_2 **DO**
9. $R[j] = A[q + j]$
10. **ENDDO**
11. $L[n_1 + 1] = \infty$
12. $R[n_2 + 1] = \infty$
13.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

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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)

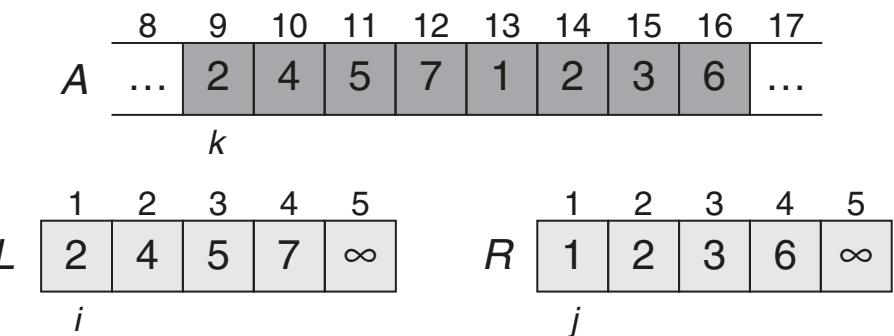
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.  ENDIF
23. ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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)

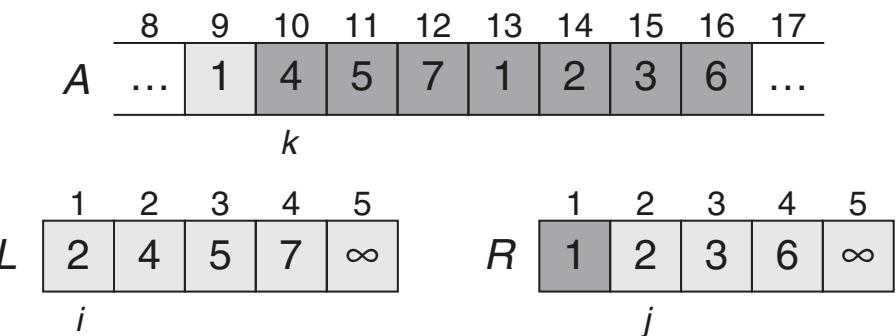
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.  ENDIF
23. ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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)

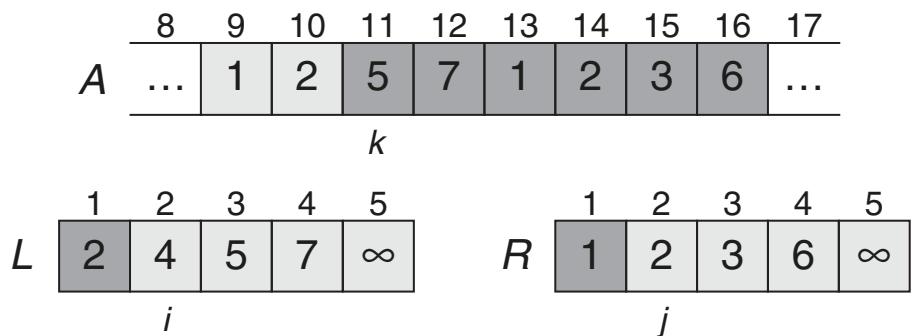
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.  ENDIF
23. ENDDO
```

Eigenschaften:

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

Beispiel:

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

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

k

L	1	2	3	4	5	
	2	4	5	7	∞	

i

R	1	2	3	4	5	
	1	2	3	6	∞	

j

Merge Sort

Merge

Merge(A, p, q, r)

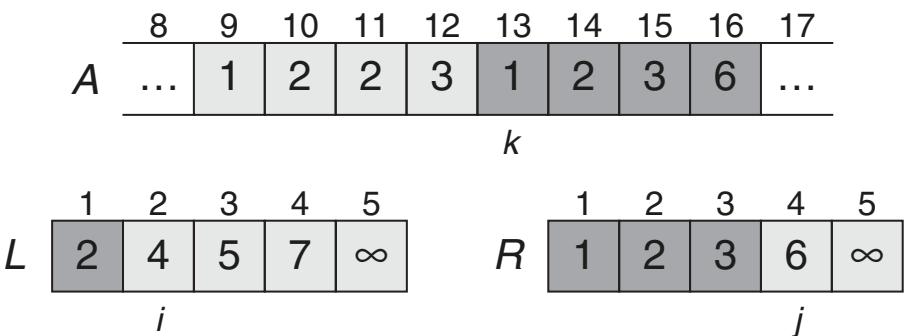
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.  ENDIF
23. ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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)

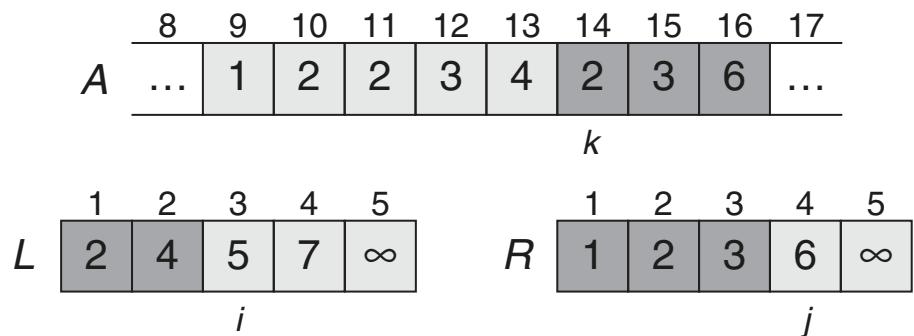
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.  ENDIF
23. ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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)

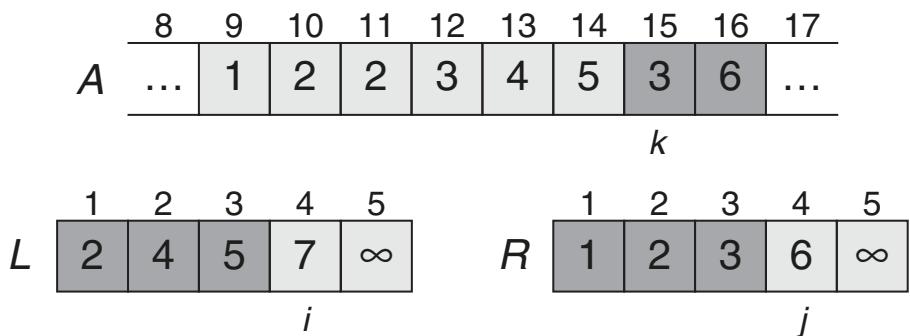
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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)

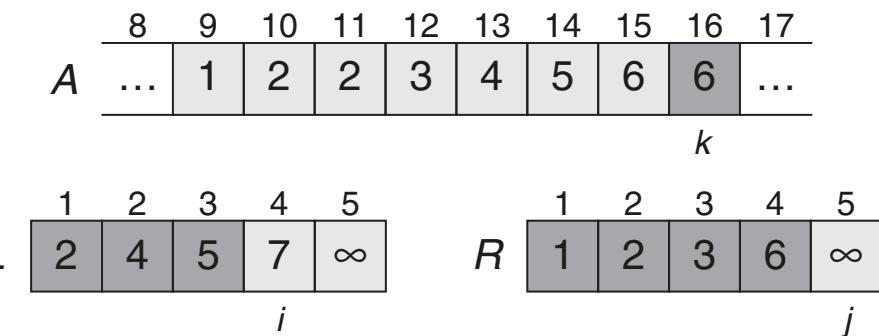
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.    ENDIF
23.  ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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)

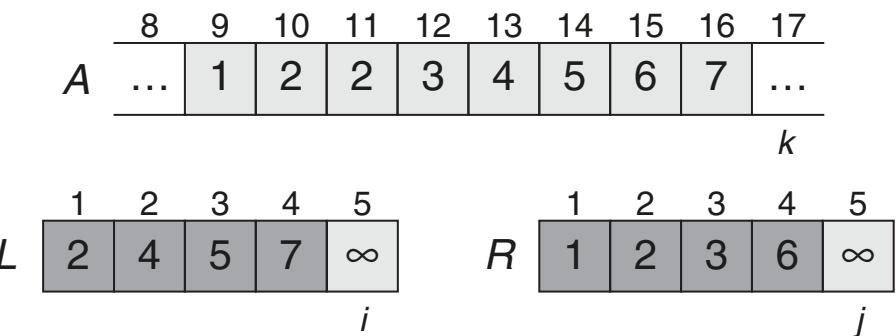
```
1.    $n_1 = q - p + 1$ 
|   ...
13.   $i = 1$ 
14.   $j = 1$ 
15.  FOR  $k = p$  TO  $r$  DO
16.    IF  $L[i] \leq R[j]$  THEN
17.       $A[k] = L[i]$ 
18.       $i = i + 1$ 
19.    ELSE
20.       $A[k] = R[j]$ 
21.       $j = j + 1$ 
22.  ENDIF
23. ENDDO
```

Eigenschaften:

- Benötigt temporären Speicherplatz
- Sentinel-Wert vermeidet Anweisungen
Hinzufügen von ∞ 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**

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:
 $n \cdot \Theta(1) = \Theta(n)$
- Gesamtlaufzeit: $\Theta(n)$.

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 $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

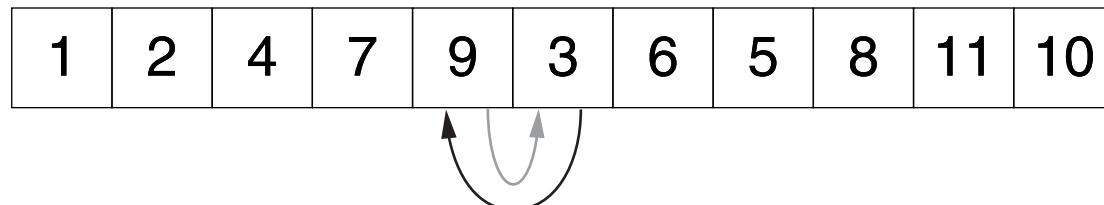
Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

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 für all $i < q$ gilt $A[i] \leq A[q]$ und für alle $j > q$ gilt $A[j] > A[q]$ 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**

Vergleiche mit Merge Sort.

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 $A[q] < A[j]$ 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:

1	2	3	4	5	6	7	8
i	p, j						

Quicksort

Partitionierung

Algorithmus: Partition.

Eingabe: A. Array von m Zahlen.

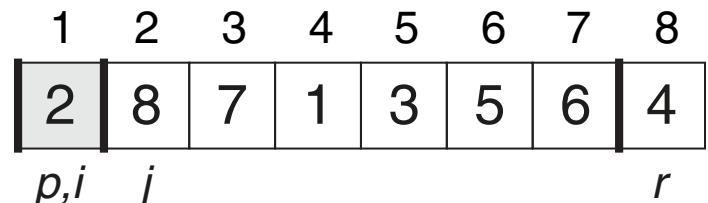
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

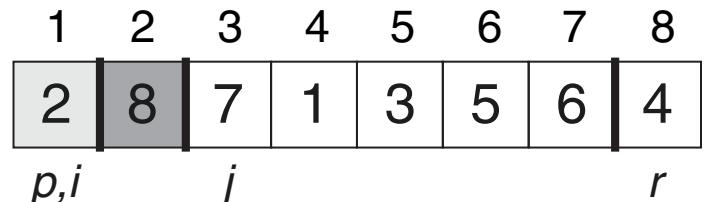
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

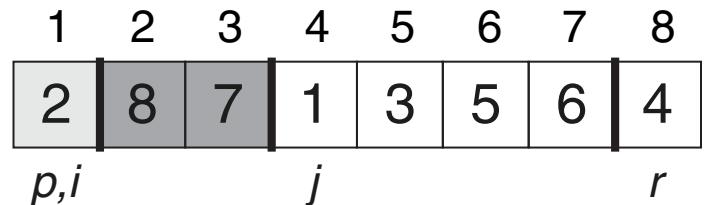
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

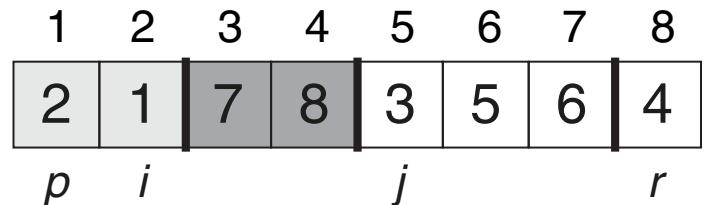
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

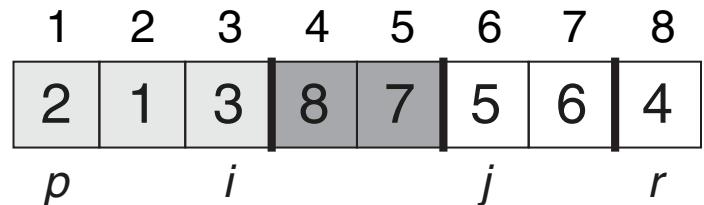
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

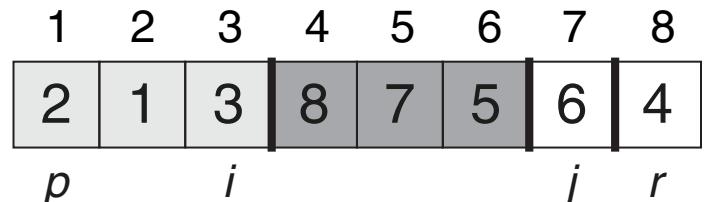
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

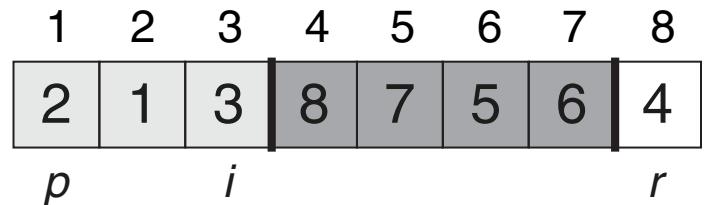
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

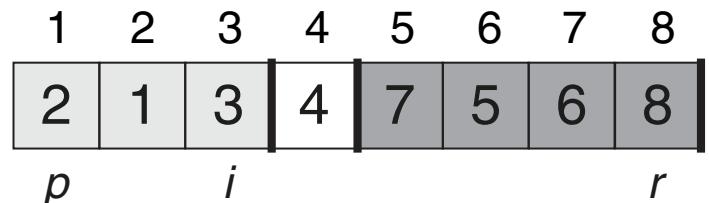
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

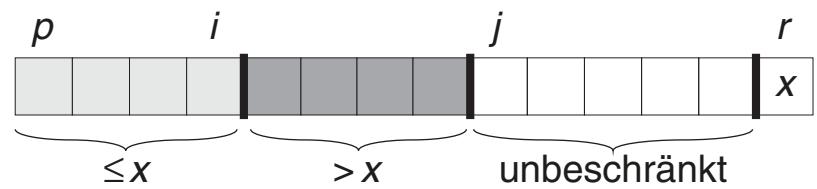
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

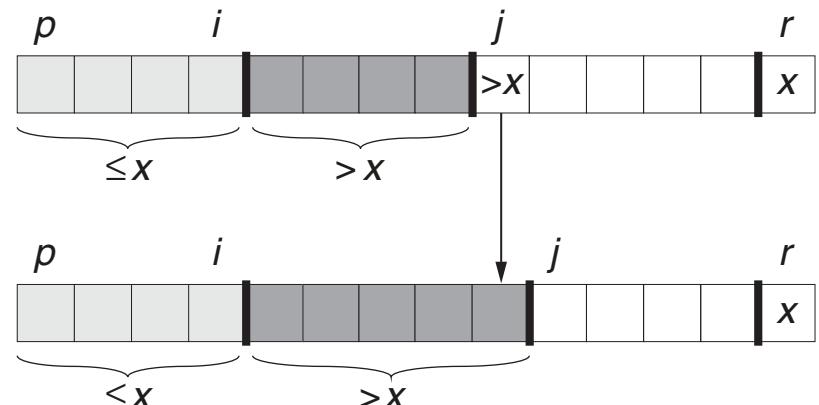
p, r . Indexe, so dass $0 < p \leq r \leq m$.

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

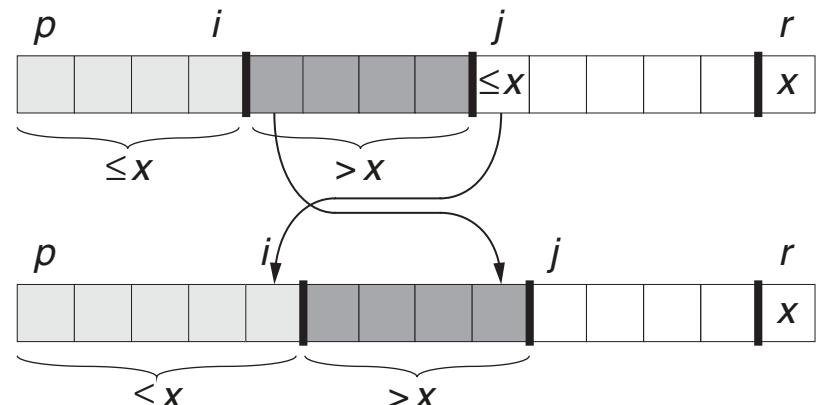
p, r . Indexe, so dass $0 < p \leq r \leq m$.

Ausgabe: Index q , so dass $p \leq q \leq r$ und $A[i] \leq A[q]$ für alle $i \in [p, q - 1]$ und $A[q] < A[j]$ 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 $A[q] < A[j]$ 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 Pivotelement.
- 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 Pivotelement.
- 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 Pivotelement mittig zwischen die beiden Partitionen getauscht, um die Vorsortierung abzuschließen.
- Dieser Partitionierungsalgorithmus wurde von Nico Lomuto vorgeschlagen.
- Charles Antony Richard Hoare, Erfinder von Quicksort, hat ein anderes Verfahren vorgeschlagen, bei dem das Pivotelement 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 Pivotelements der Median aller Werte in A ist.

Quicksort

Laufzeit (informell)

Statische Partitionierung:

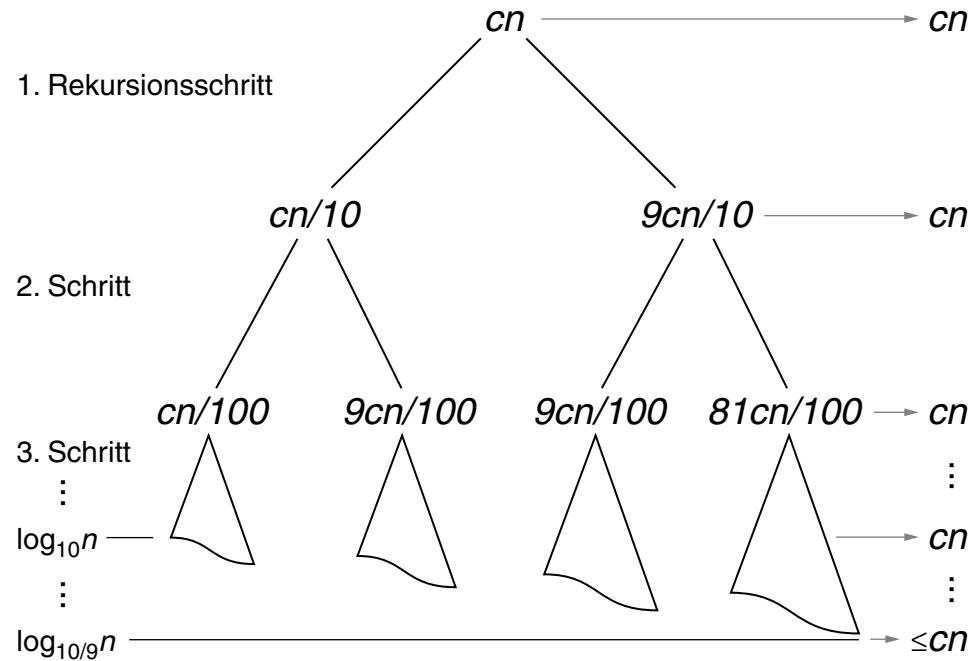
- Immer eine 9:1-Partitionierung: 90% der Elemente vs. 10% der Elemente.
- $T(n) = T(n/10) + T(9n/10) + \Theta(n)$

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

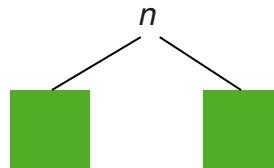
Laufzeit (informell)

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

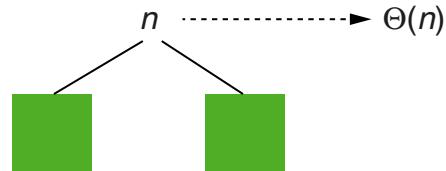
Laufzeit (informell)

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

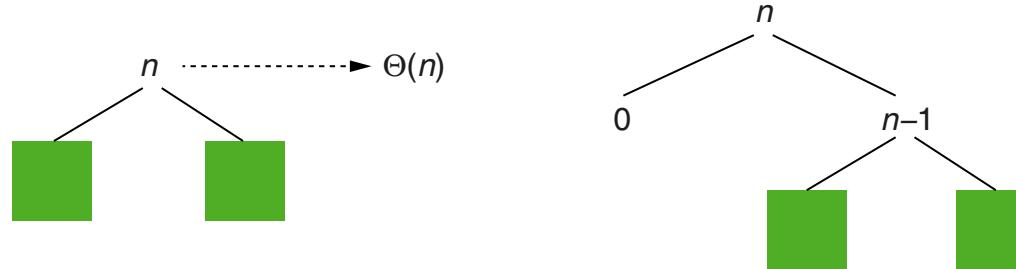
Laufzeit (informell)

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

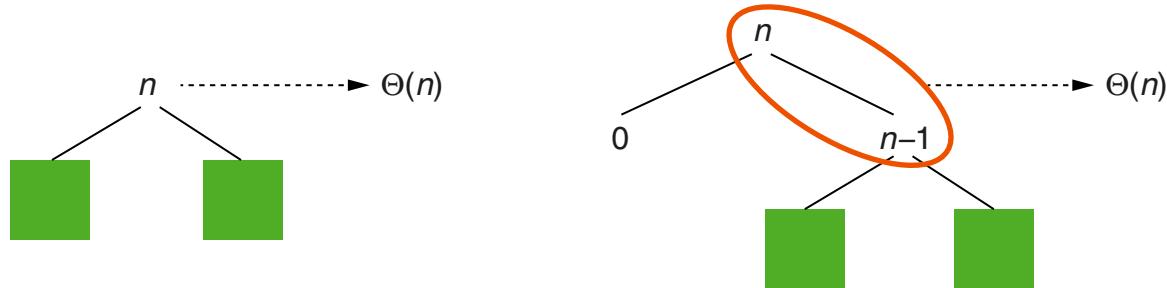
Laufzeit (informell)

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 Pivotelement ab:

- **Statisches Element**

Ein vordefiniertes Element wird als Pivotelement 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 Pivotelement. 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 Pivotelement zunächst an die richtige Position getauscht werden.

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