11 Weitere Sortieralgorithmen

Quicksort

Sortierverfahren

- rekursiver, nicht-stabiler Sortieralgorithmus
- arbeitet nach „Teile und Beherrsche“
- Sortierung erfolgt meist sehr speichereffizient „in place“

Quicksort

Divide and Conquer

- Divide: Wähle ein Pivot-Element p, das das Array A[l...r] in zwei Teilarrays teilt.
  - Sortiere kleinere Elemente in die linke Teilliste:
    A[l...p-1] mit $a \leq A[p]$, alle $a \in A[l...p-1]$
  - Sortiere größere Elemente in die rechte Teilliste
    A[p+1...r] mit $a \geq A[p]$, alle $a \in A[p+1...r]$
  - Element A[p] steht nun bereits an der richtigen Position.

- Conquer: Sortiere A[l...p-1] und A[p+1...r] durch rekursiven Aufruf.
- Merge: Da „in place“ sortiert wird, ist kein Zusatzaufwand nötig.

Quicksort

Algorithmus I

```
1.  function QUICKSORT(A, l, r)
2.      if l < r then
3.          p = PARTITION(A, l, r)
4.          QUICKSORT(A, l, p-1)
5.          QUICKSORT(A, p+1, r)
6.      end if
7.  end function
```

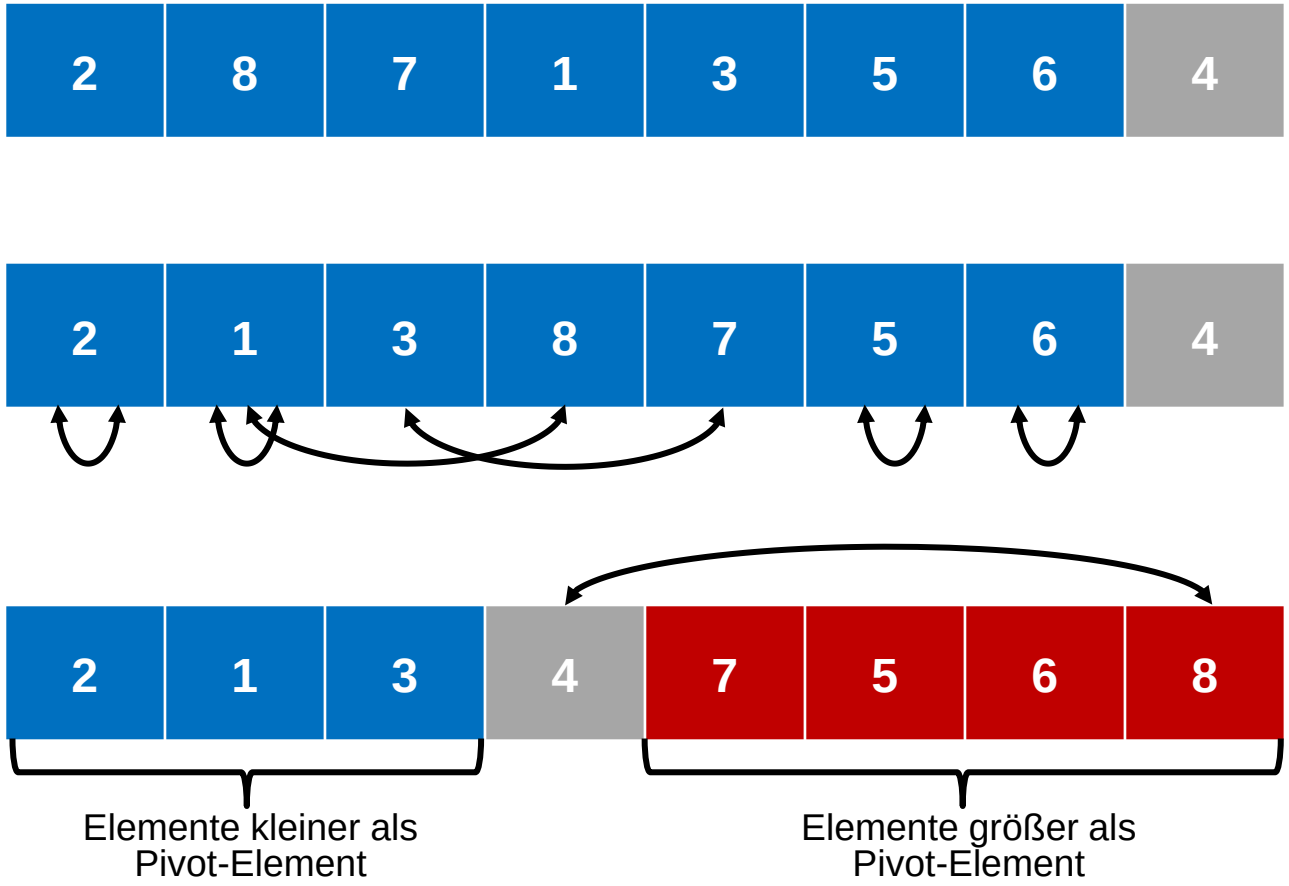
Quicksort

Algorithmus II

```
1.  function PARTITION(A, l, r)
2.      x = A[r]
3.      i = l -1
4.      for j =l to r – 1 do
5.          if A[j] ≤ x then
6.              i = i + 1
7.              A[i] ⟷ A[j]
8.          end if
9.      end for
10.     A[i+1] ⟷ A[r]
11.     return i + 1
12. end function
```

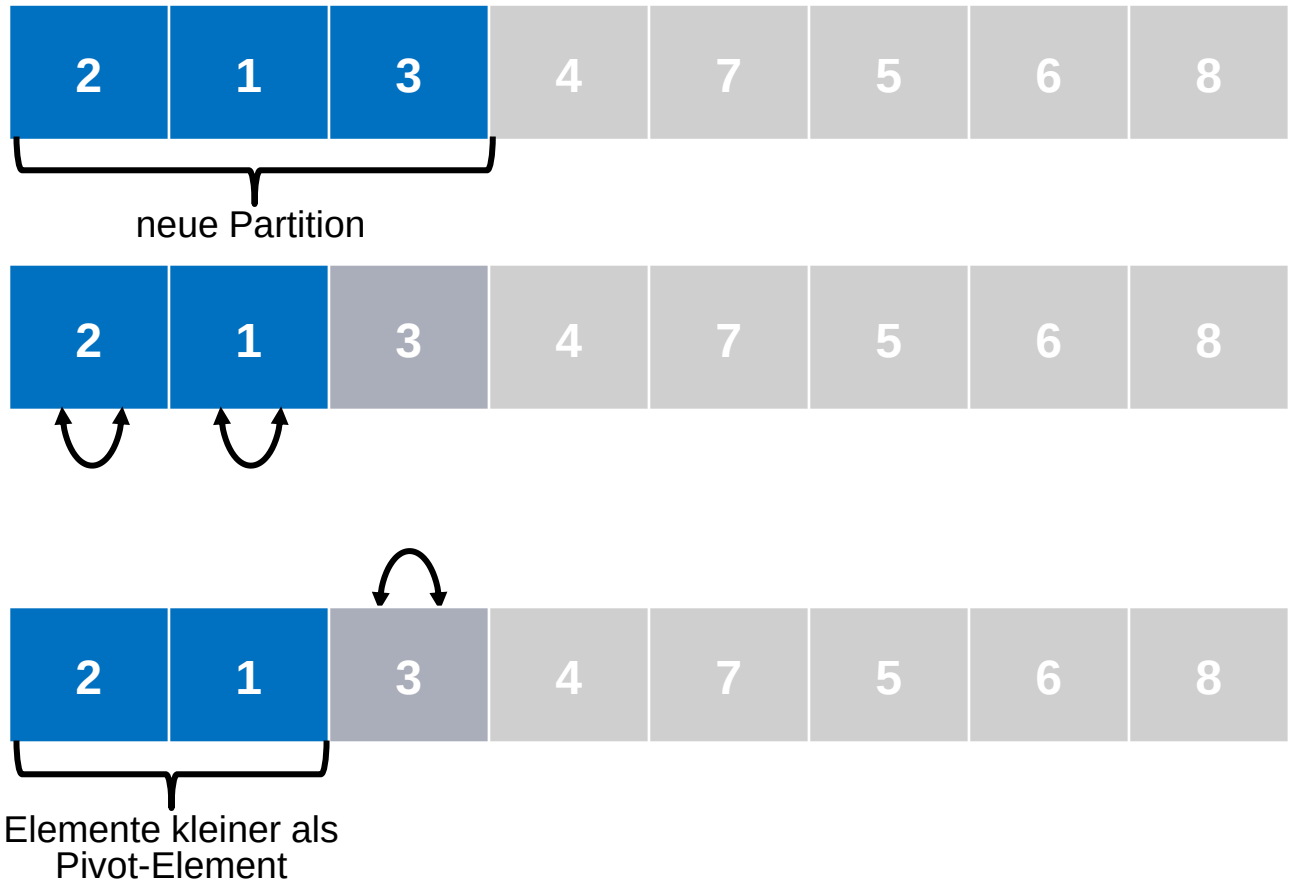
Quicksort

Beispiel 1: PARTITION, 1. Aufruf, Rückgabe = Index 4



Quicksort

Beispiel 1: PARTITION, 2. Aufruf, Rückgabe = Index 3

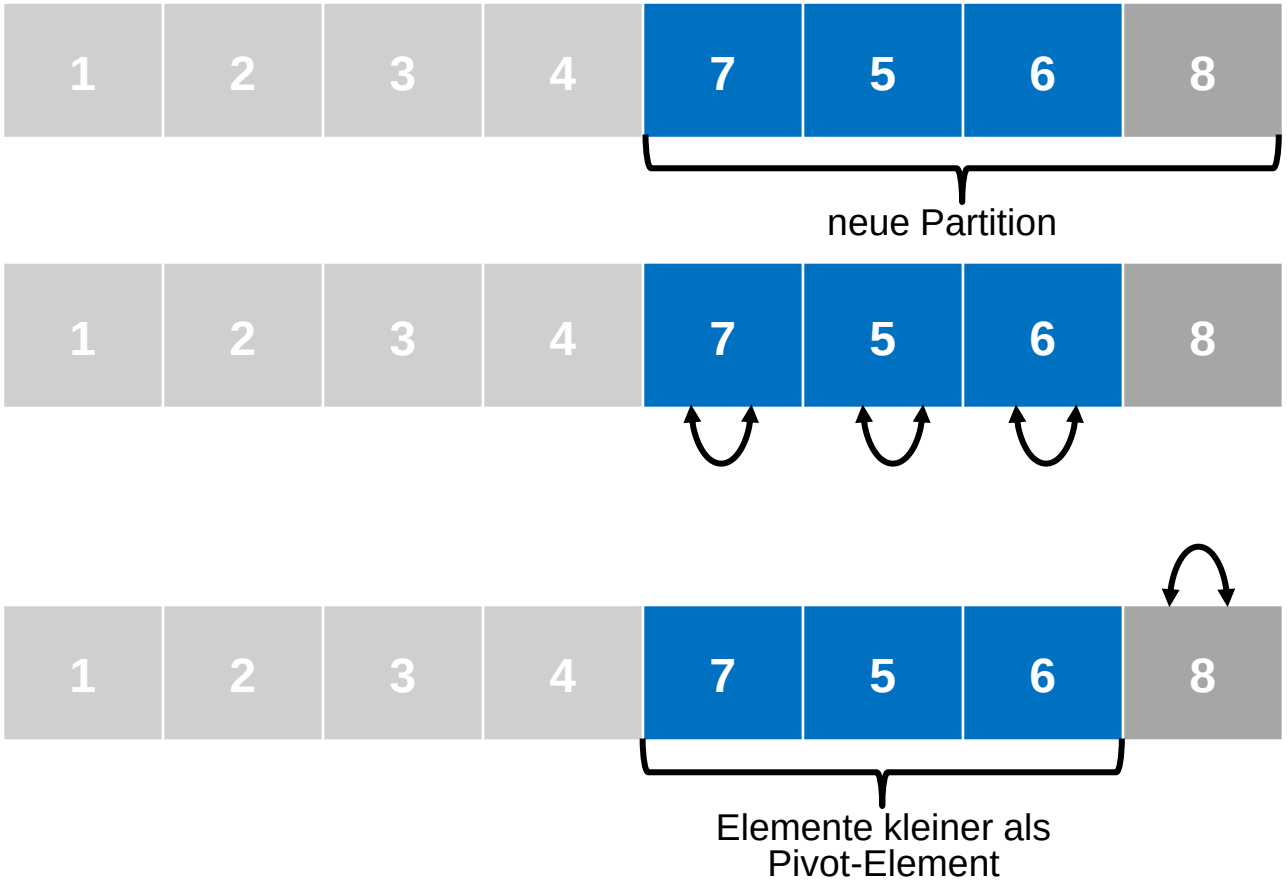


Quicksort

Beispiel 1: PARTITION, 3. Aufruf, Rückgabe = Index 1

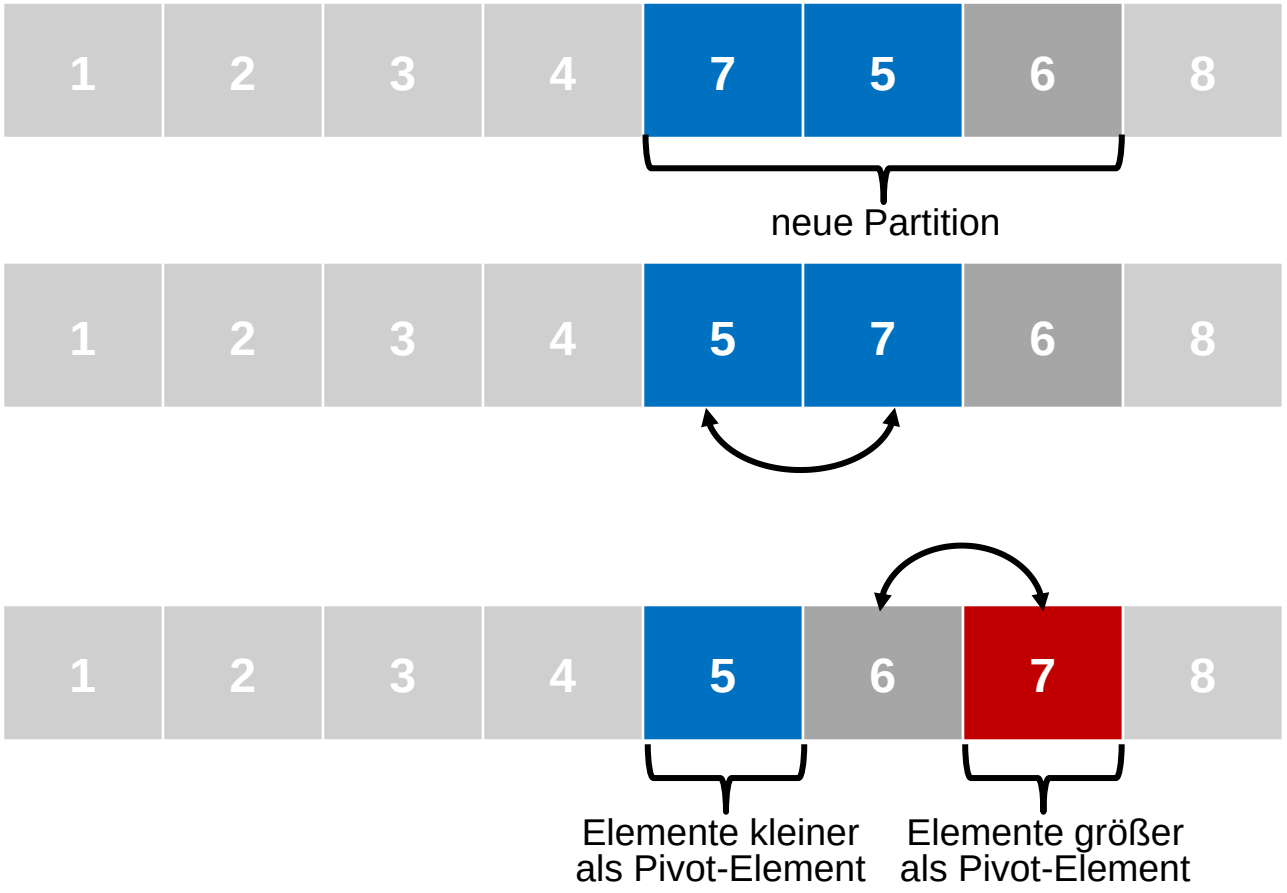
Quicksort

Beispiel 1: PARTITION, 4. Aufruf, Rückgabe = Index 8



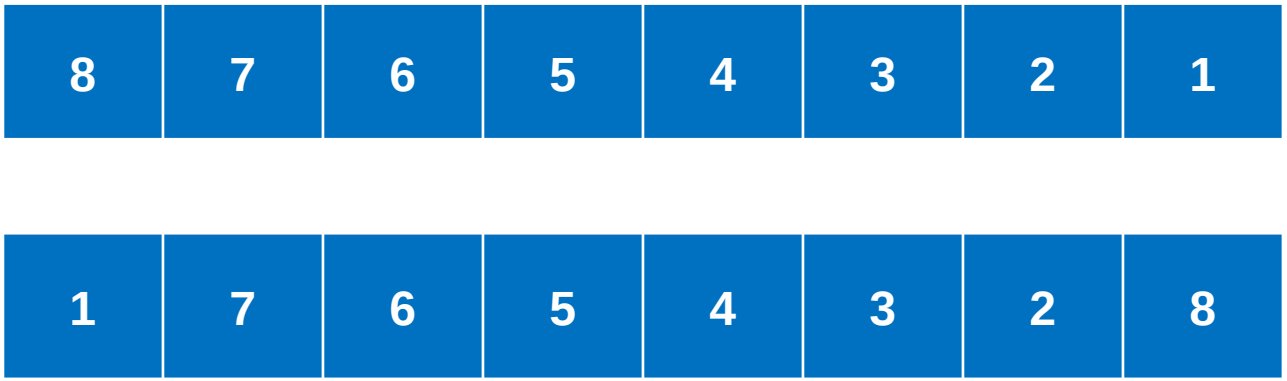
Quicksort

Beispiel 1: PARTITION, 5. Aufruf, Rückgabe = Index 6



Quicksort

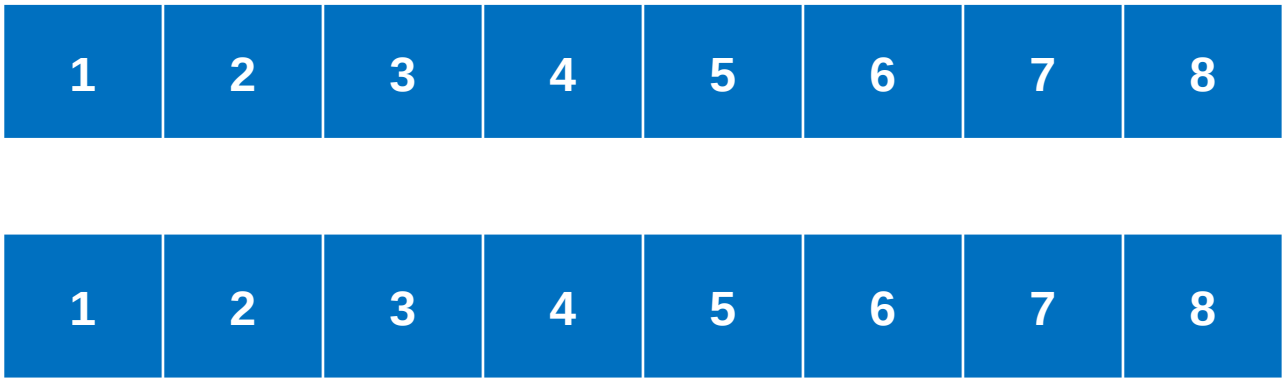
Beispiel 2: PARTITION



Welches p liefert die erste Partitionierung? 1

Quicksort

Beispiel 3: PARTITION



Welches p liefert die erste Partitionierung? 8

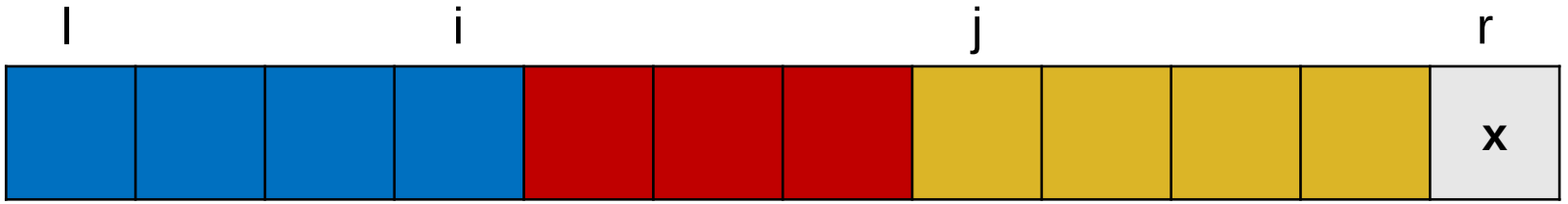
Quicksort

Beobachtungen

- Wert von A[r] (Pivotelement) bestimmt wesentlich das Ergebnis
- Genauer: Position des Pivotelements in der sortierten Folge bestimmt wesentlich das Ergebnis
- Günstig ist eine jeweils gleich große Aufteilung
  → jeweils Halbierung des Problems
- Partitionierung bestimmt Laufzeit

Quicksort

Partitionierung



Farben

- Blau: Werte ≤ x
- Rot: Werte > x
- Gelb: Werte noch unbearbeitet

Quicksort

Korrektheit

Schleifeninvariante:

1. $A[k] \leq x$ $\quad l \leq k \leq i$
2. $A[k] > x$ $\quad i + 1 \leq k \leq j - 1$
3. $A[k] = x$ $\quad k = r$

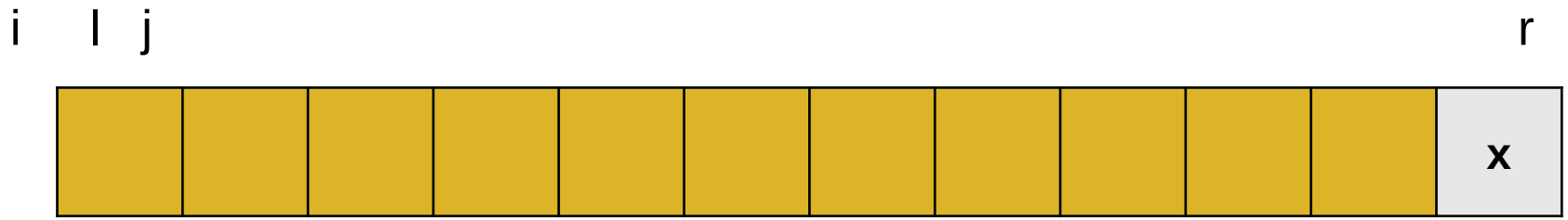
Quicksort

Korrektheit

Initialisierung:

- i = l – 1, j = l
- Es liegen keine Werte zwischen l und i und zwischen i + 1 und j – 1
- Durch Zeile 2 im Algorithmus ist Bed. 3 erfüllt.

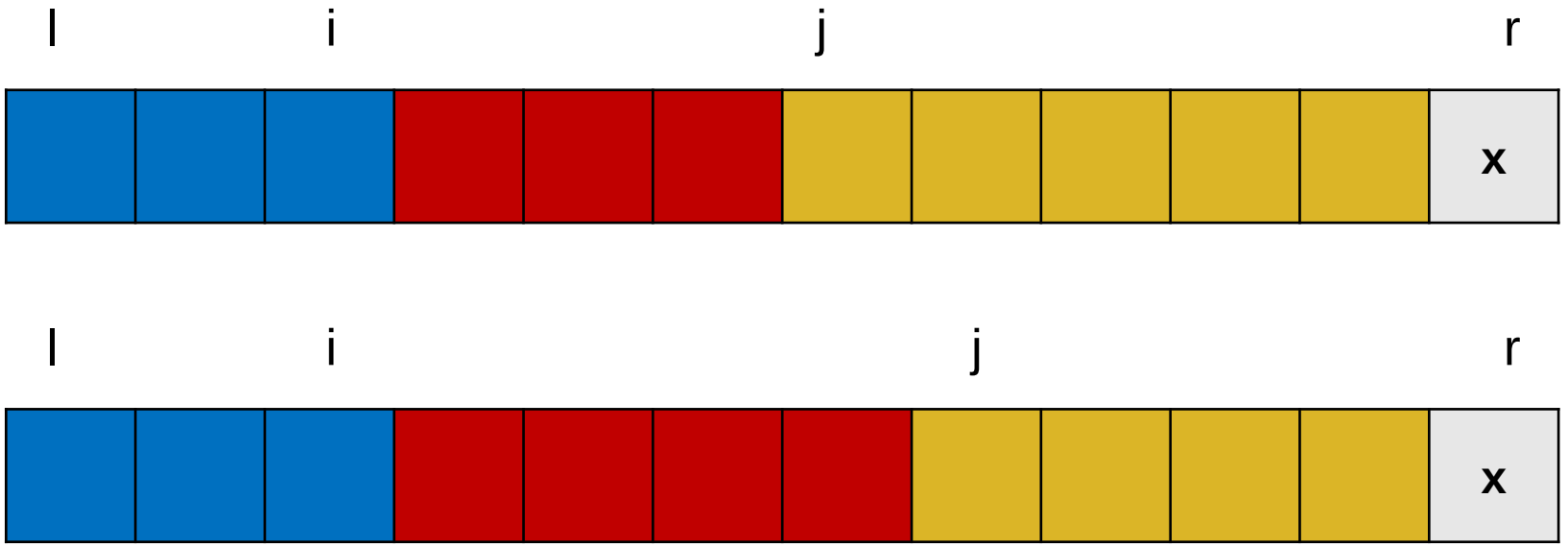
  → Schleifeninvariante erfüllt



Quicksort

Korrektheit

Aufrechterhaltung: Fall A[j] > x

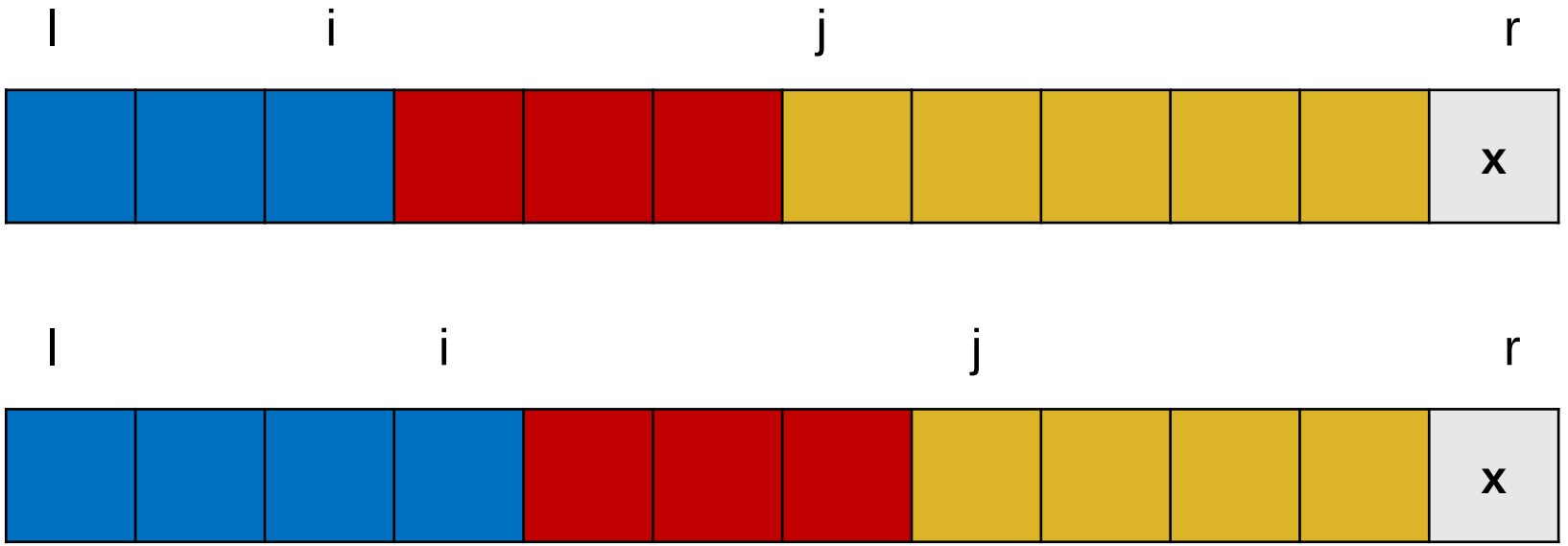


→ Schleifeninvariante erfüllt

Quicksort

Korrektheit

Aufrechterhaltung: Fall A[j] > x

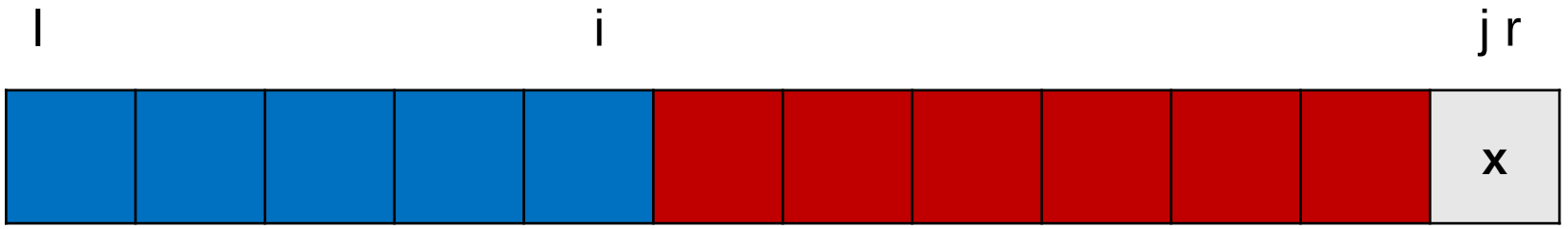


→ Schleifeninvariante erfüllt

Quicksort

Korrektheit

Terminierung



- j = r Jedes Arrayelement ist in einer der 3 Partitionierungen
- Erst danach Vertauschung A[i+1] ↔ A[r]

→ Schleifeninvariante erfüllt

Quicksort

Laufzeit

Worst Case

Partitionierung liefert ein Teilarray mit 0 Elementen und eines mit n-1 Elementen

$$T(n) = T(0) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n)$$

Wiederholtes Einsetzen liefert

$$T(n) = \Theta(n) + \Theta(n-1) + \cdots + \Theta(1) = \Theta(\sum_{i=1}^{n} i)\Theta(n)$$

Es gilt also: $T(n) = \Theta(n^2)$

Dieser Fall tritt auf, wenn das Array A bereits sortiert ist.

Quicksort

Laufzeit

Best Case

Partitionierung teilt immer in zwei gleich große Hälften

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

Lösungsmöglichkeiten:

1. Analyse des Rekursionsbaums
2. Mastertheorem

Quicksort

Laufzeit

Best Case (Rekursionsbaum)

Rekursionsbaum ist im Idealfall vollständig ausgewogener Binärbaum mit $log_2(n)$ Schichten mit einem Aufwand von jeweils $\Theta(n)$.

Gesamtaufwand ist also: $\mathrm{T(n)} = \Theta(nlog_2 n)$

Quicksort

Laufzeit

Best Case

Etwas allgemeiner kann gezeigt werden, dass jede balancierte Partitionierung mit einem konstanten Verhältnis asymptotisch so gut ist, wie die Partitionierung im besten Fall.

Quicksort

Laufzeit

Average Case

Im mittleren Fall gilt:

$$\mathrm{T(n)} = O(nlog_2 n)$$

Quicksort

Laufzeit

Zusammenfassung

- Worst case: (sehr unwahrscheinlich)
- Best case: (relativ unwahrscheinlich)
- Average case: (fast immer)

Diese Betrachtungen machen den Quicksort zu einem der besten Sortierverfahren.

Auswahl des Pivotelements

Auswahl des Pivotelements ist für die Laufzeit entscheidend ist. Alternativen

1. Wähle das erste oder letzte Element des zu sortierenden Teilarrays, d.h. x = A[l] oder x = A[r]
2. Wähle zufällig ein Element (Randomized Quicksort)
3. Wähle das mittlere Element aus den Elementen A[l], A[r] und A[(l+r)/2] (bzgl. der Relation $\leq$). Diese Variante hat sich in der Praxis bewährt.

Quicksort: Weitere Überlegungen

Randomisierte Partitionierung

1. function RANDOMIZED-PARTITION(A, l, r)
2. i = RANDOM(l,r)
3. exchange A[r] ↔ A[i]
4. return PARTITION(A,l,r)
5. end function

Quicksort: Weitere Überlegungen

Randomisierter Quicksort

```
1. function RANDOMIZED-QUICKSORT(A, l, r)
2.     if l < r then
3.         q = RANDOMIZED-PARTITION(A, l, r)
4.         RANDOMIZED-PARTITION(A, l, p-1)
5.         RANDOMIZED-PARTITION(A, p+1, r)
6.     end if
7. end function
```

Bogosort

Sortierverfahren

- auch Monkeysort oder Stupidsort
- rekursiver, nicht-stabiler Sortieralgorithmus
- Vertausche die Zahlen zufällig, solange bis sie sortiert sind.

Bogosort

Algorithmus

1. function BOGOSORT(A)
2. if A is not sorted then
3. SHUFFLE(A)
4. end if
5. end function

Bogosort

Laufzeiten

- $T_{ac} \in O(n \cdot n!)\ bzw.\ \infty$