Informatik 1

Sortieren

Inhalt

- Invariante
- Sortieren
  - Direkte Auswahl
  - Direktes Einfügen
  - Rekursives Mergesort
  - Quicksort
- Weitere Sortierverfahren
- Untere Schranke

Invariante (1/5)

- Vorbedingung: Logische Aussage, die vor einem Programmabschnitt gilt.
- Nachbedingung: Logische Aussage, die nach einem Programmabschnitt gilt.
- Invariante: Logische Aussage über ein Programmabschnitt, der immer gilt (für jede gegebene Eingabe des Problems)
- Schleifeninvariante: Invariante, die vor einer Schleife und für jeden Schleifendurchlauf immer gilt.
- Geeignete Invarianten beschreiben die Korrektheit eines Verfahrens. Sie helfen, einen Algorithmus abstrakt zu verstehen.
- Die Aussage „true“ ist immer eine Vor-, Nachbedingung oder Invariante. Aber normalerweise keine geeignete.
- z.B. zur Fehlererkennung in Java programmierbar mit
  ```
  assert BoolescherAusdruck;
  ```
  Sie werden zur Laufzeit überprüft (abschaltbar)

Invariante (2/5)

- Beispiel: Person und heiraten(…)
  - Gültigkeit der Vorbedingung folgt aus der Implementierung von darfheiraten()
  - Gültigkeit der Nachbedingung folgt aus Vorbedingung und dem Programmabschnitt

```
public void heiraten(Person ehegatte) {
      if ( this.darfHeiraten(ehegatte) ) {
         // Vorbedingung:
         // this und ehegatte sind verschiedene, volljährige
         // und unverheiratete Personen
         this.name = this.name + "-" + ehegatte.name;
         ehegatte.name = this.name;
         this.ehegatte = ehegatte;
         ehegatte.ehegatte = this;
         // Nachbedingung:
         // this und ehegatte sind miteinander verheiratet
      }
   }
```

Invariante (3/5)

- Beispiel Invariante: Person und heiraten(…)
  - Alter ändert sich nicht durch Programmabschnitt

```
public void heiraten(Person ehegatte) {
      if ( this.darfHeiraten(ehegatte) ) {
         // Invariante:
         // this und ehegatte sind volljährig
         this.name = this.name + "-" + ehegatte.name;
         ehegatte.name = this.name;
         this.ehegatte = ehegatte;
         ehegatte.ehegatte = this;
       }
   }
```

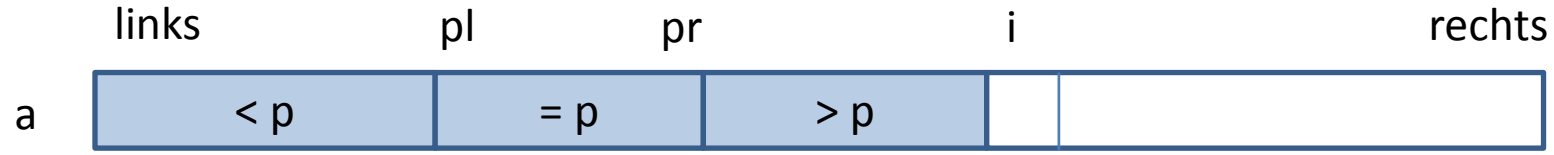
Invariante (4/5)

- Beispiel Schleifeninvariante: Euklidischer Algorithmus
- Schleifeninvariante gilt
  - vor Eintritt der Schleife (offensichtlich wegen if)
  - nach jedem Durchlauf, da die Differenz der gröseren positiven Zahl und der kleineren positiven Zahl wieder positiv ist

```
if (a > 0 & b > 0) {
  // Schleifeninvariante: a > 0 und b > 0
  while ( a != b ) {
    if (a < b) {
      b = b - a;
    } else {
      a = a - b;
    }
  }
}
```

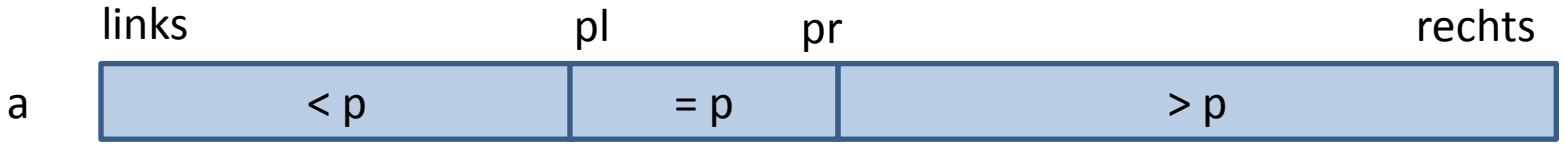
Invariante (5/5)

- Invarianten lassen sich oft **grafisch** veranschaulichen
- Beispiel: Unterteilung (Partitionierung) bei k-kleinste Element suchen
- Schleifeninvariante:



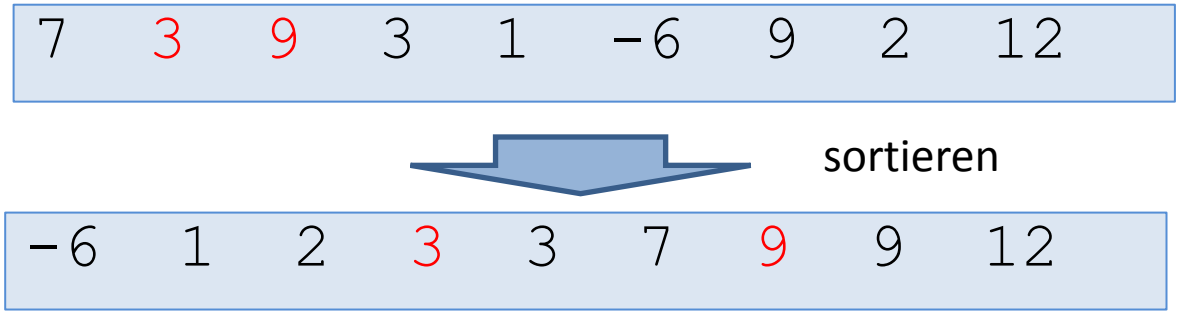
- Vorbedingung der Schleife:

- Nachbedingung der Schleife:



Sortierproblem

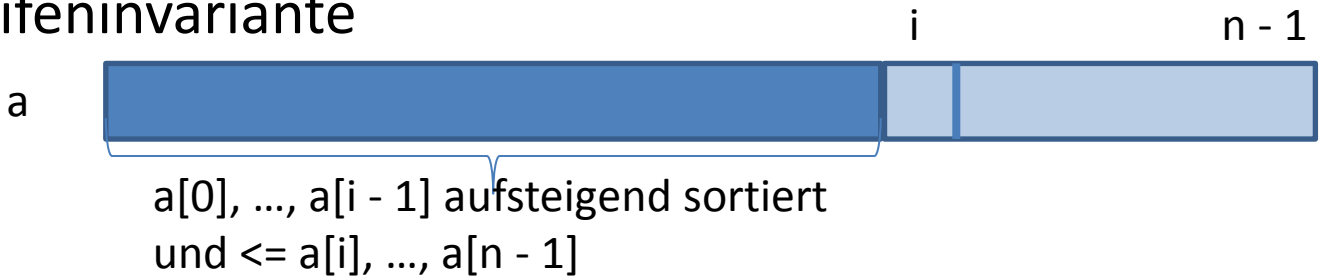
Sortierproblem	
Gegeben:	Folge a von n Werten
Gesucht :	Aufsteigend sortierte Folge mit den ursprünglichen Werten von a



- Die Sortierverfahren im folgenden sortieren die Werte von a durch Vertauschung.
- Die Eingabe a wird geändert.
- Die Eingabe a ist auch die Ausgabe.
- Es gibt keine spezielle Ausgabefolge.
- Ein Sortierverfahren heißt stabil, wenn bei gleichen Werten dessen ursprüngliche Reihenfolge in der sortieren Folge beibehalten wird.

Sortieren durch direktes Auswählen

- **Algorithmus**: Für alle i = 0 bis n: das Minimum von a[i] bis a[n - 1] suchen und mit a[i] tauschen
- stabil, wenn mit erstem Minimum von links getauscht wird
- Schleifeninvariante

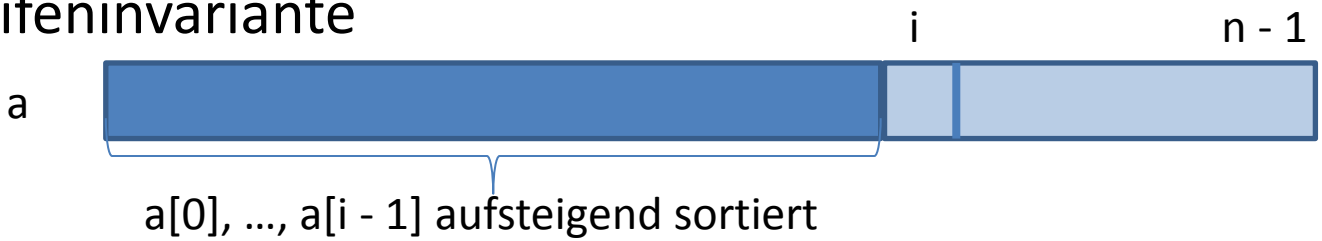


- Vorbedingung

- Nachbedingung:

Sortieren durch direktes Einfügen

- **Algorithmus**: Für alle i = 1 bis n- 1: Füge a[i] an die *richtige* Stelle in a[0] … a[i - 1] ein.
- stabil, wenn nach doppeltem Vorkommen eingefügt wird
- Schleifeninvariante

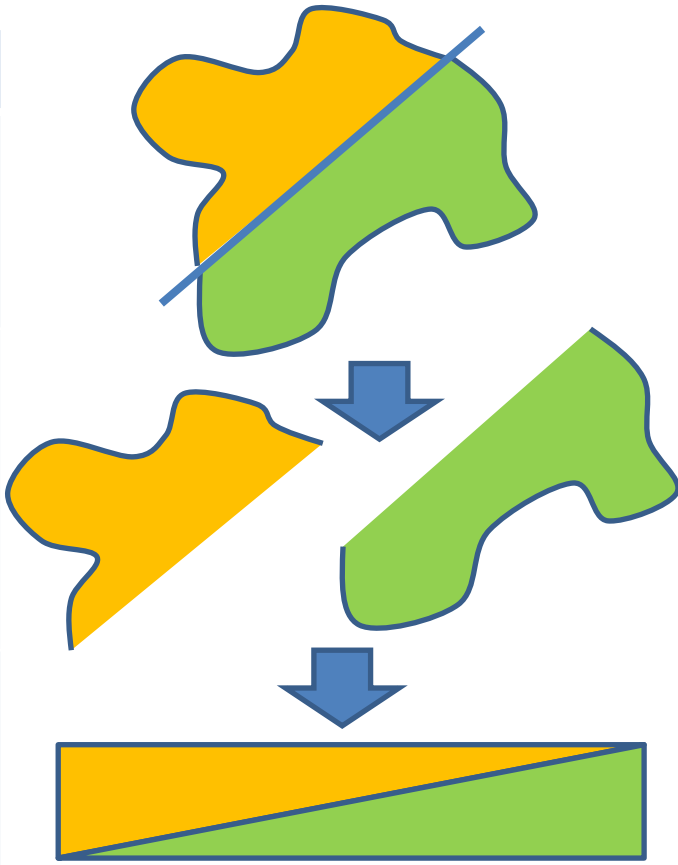


- Vorbedingung

- Nachbedingung:

Teile-und-Beherrsche-Verfahren

Algorithmus	Bezeichnung
1. Problem in mindestens zwei Teilprobleme aufteilen	Teile-Schritt
2. Teilprobleme rekursiv lösen Kleine Probleme direkt lösen (Rekursionsabbruch)	
3. Lösung aus Teillösungen konstruieren	Beherrsche-Schritt



Halbierungsverfahren ist ein **Spezialfall** eines Teile-und-Beherrsche-Verfahrens. Teile-und-Beherrsche-Verfahren sind ähnlich zu Dynamischen Programmieren, aber die Teillösung werden nicht dauerhaft zwischengespeichert.

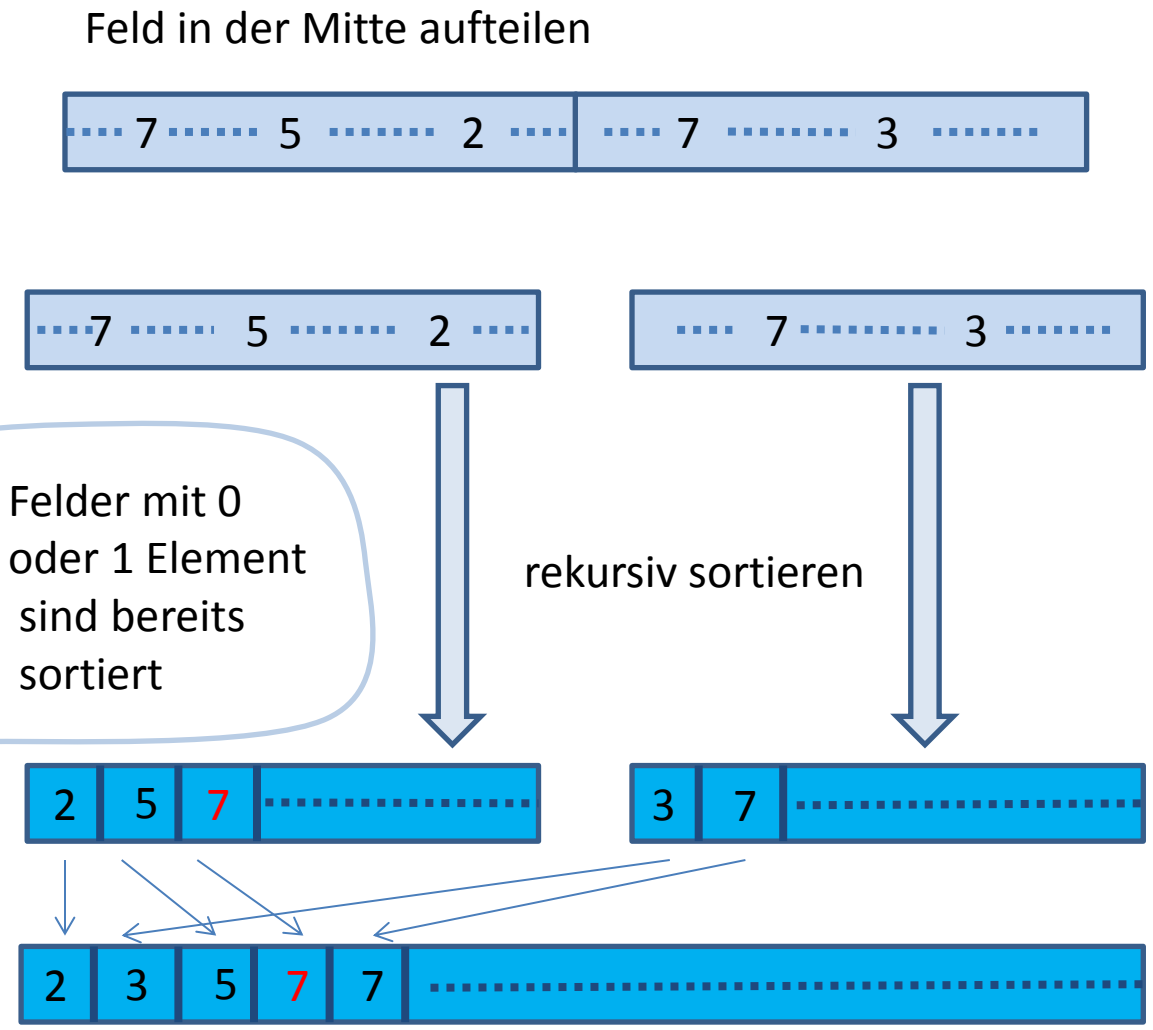
Mergesort (1/3)

1. Problem in der Mitte aufteilen

2. Teilprobleme rekursiv sortieren

Kleine Probleme direkt lösen (Rekursionsabbruch)

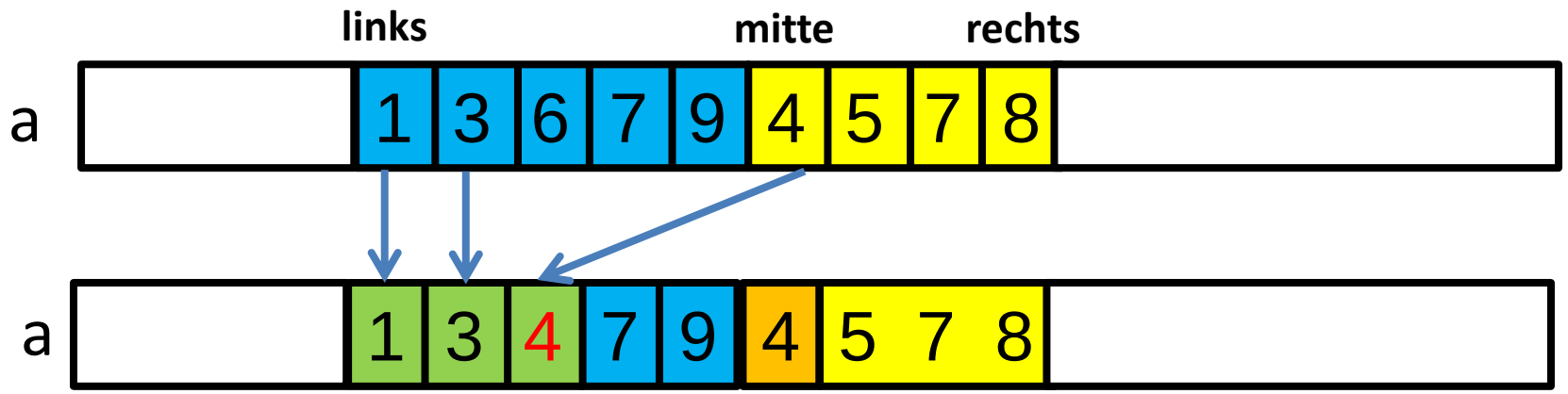
3. Sortierte Teilfolgen zu sortierten Gesamtfolge **verschmelzen** (to merge)



stabil, wenn bei Gleichheit der Wert von linker Folge zuerst kopiert wird

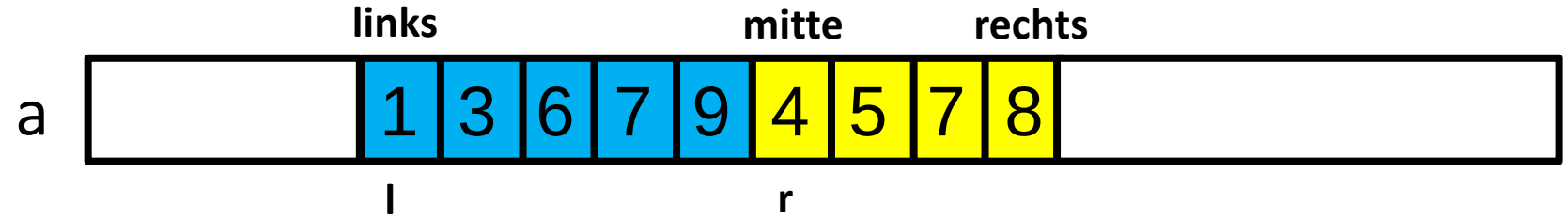
Mergesort (2/3)

```
private void sortieren(int [] a, int links, int rechts) {
  if (rechts > links) {
    int mitte = (links + rechts) / 2;
    sortieren(a, links, mitte - 1);
    sortieren(a, mitte, rechts);
    verschmelzen(a, links, mitte, rechts);
  }
}
```

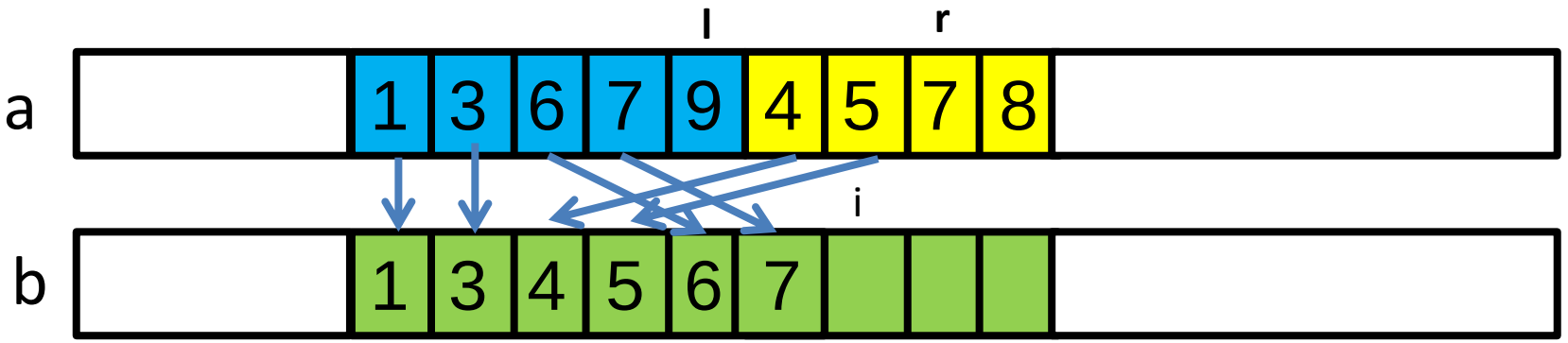


Verschmelzen benötigt zusätzlichen Speicher b

Mergesort (3/3)



- i = links bis rechts hochzählen und kleinsten Wert von a[l] und a[r] nach a[i] kopieren, l bzw. r hochzählen



- Zum Schluss einer Schleife wieder zurück nach a kopieren
- Zusätzliches Feld b nur einmal erzeugen!

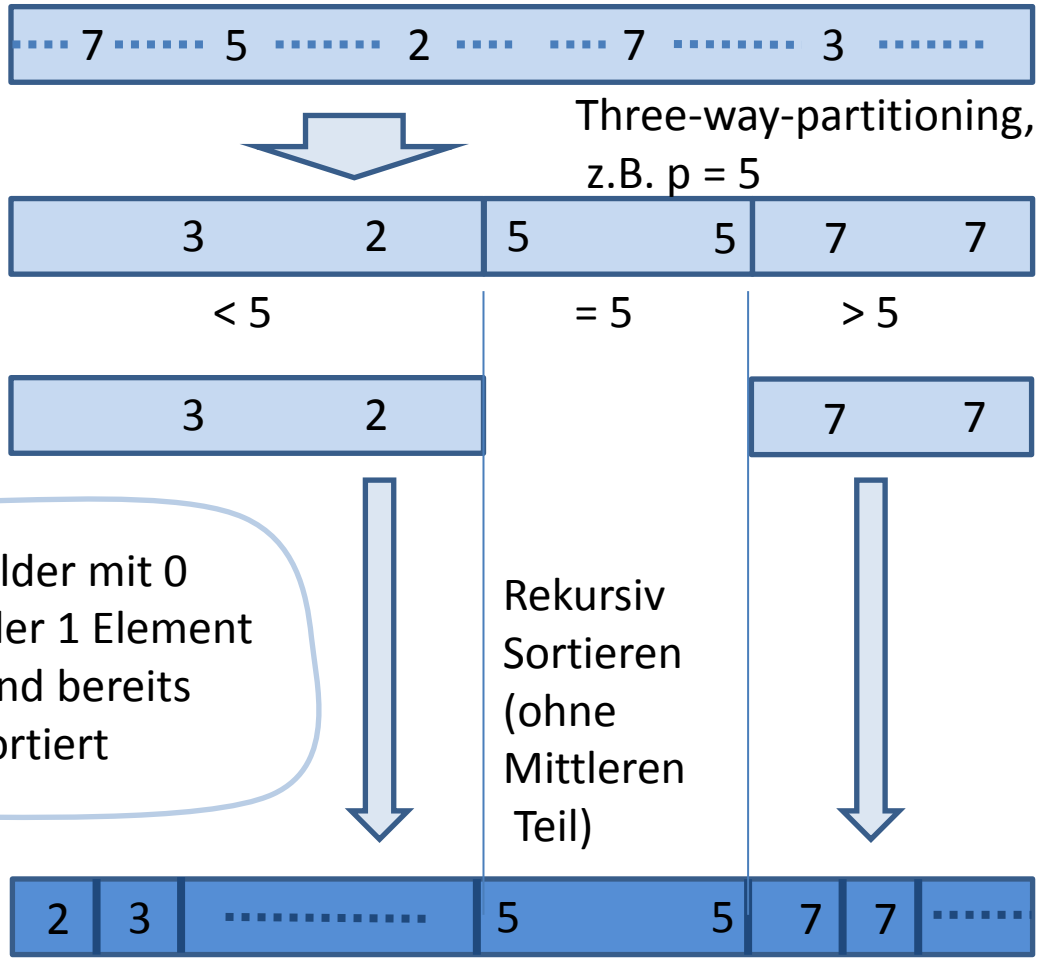
Quicksort (1/5)

1. Problem unterteilen mit einem Auswahlelement p

2. Teilprobleme rekursiv sortieren

Kleine Probleme direkt lösen (Rekursionsabbruch)

3. Nichts zu tun



Nicht stabil z.B. bei a = 5 6 7 6 5, p = 5. Partitionierung 5|6 7 6 | 5, letzte a[i] = 5, 5 mit 6 vorne vertauscht ergibt 5 5 | 7 6 6 |

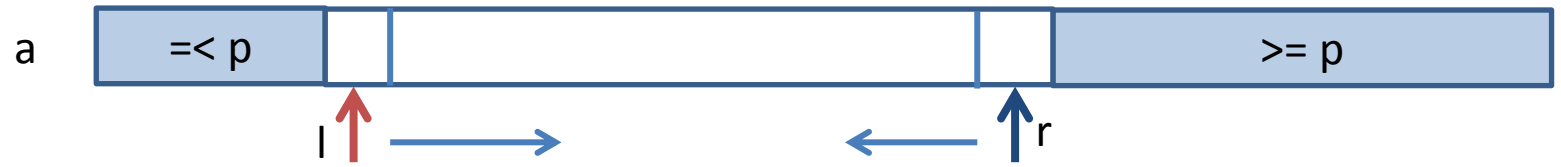
Quicksort (2/5)

```
private void sortieren(int [] a, int links, int rechts) {
   if (links < rechts) {
      int p = a[links];
      int pl = links;
      int pr = links;
      for (int i = links + 1; i <= rechts; i++) {
         if (a[i] == p) {
              a[i] = a[pr++];
              a[pr] = p;  // pr++ vorgezogen, um Zeilen zu sparen
         } else if (a[i] < p) {
              a[pl++] = a[i]; // pl++, pr++ vorgezogen
              a[i] = a[++pr];
              a[pr] = p;
         }
      }
     sortieren(a, links, pl - 1);
     sortieren(a, pr + 1, rechts);
   }
}
```

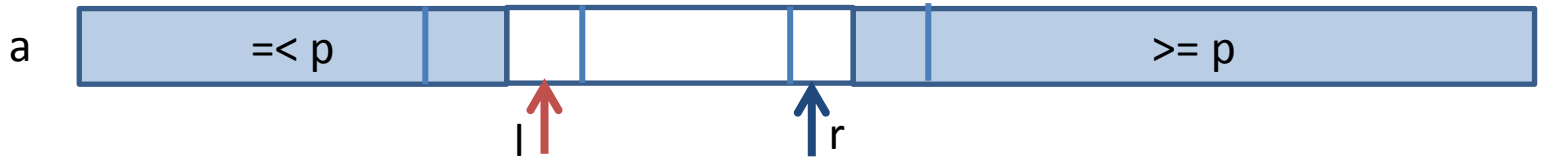
Quicksort (3/5)

- Alternative: **Two-way-partitioning**

  - Vorteil: Weniger Vertauschungen nötig
  - Nachteil: bei vielen doppelten Werten mehr rekursive Aufrufe
  - Schleifeninvariante für Partitionierung:



- Solange beide Pfeile nicht überschneiden:
  - Roten Pfeil mit Index l solange nach rechts bewegen bis a[l] >= p ist,
  - blauen Pfeil mit index r solange nach links bewegen bis a[r] <= p ist,
  - falls l <= r gilt: a[l] mit a[r] vertauschen, l++, r-- .



Quicksort (4/5)

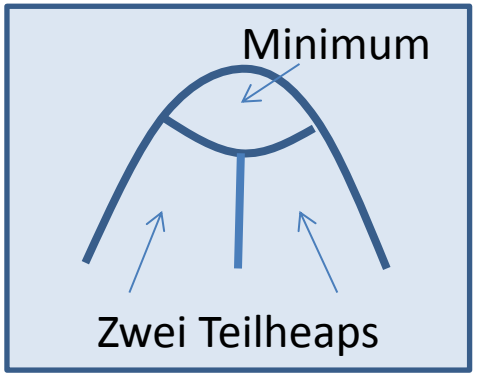
```
private void sortieren(int [] a, int links, int rechts) {
if (links < rechts) {
  int p = a[(links + rechts) / 2];
  int l = links;
  int r = rechts;
  while (l <= r) {
    while (a[l] < p) l++; // Block weggelassen, um
    while (a[r] > p)  r--; // Zeilen zu sparen
    if (l <= r) {
      int t = a[l];
      a[l] = a[r];
      a[r] = t;
      l++;
      r--;
    }
  }
  sortieren(a, links, r);
  sortieren(a, l, rechts);
}
```

Quicksort (5/5)

- Quicksort ist sehr schnell, außer im (sehr seltenen) schlechten Fall
- Hybride Version
  - Rekursion bei kleinem n, z. B. n <= 25, abbrechen und Direktes Einfügen verwenden
    - Konstanter Faktor reduziert sich dadurch
  - Nach der Partitionierung schlechten Fall erkennen: in O(1) Schritten möglich
    - In diesem Fall in der Mitte wie bei Mergesort teilen
    - Beiden Hälften rekursiv sortieren
    - Danach die Hälften verschmelzen
  - Konstanter Faktor erhöht sich dadurch nur unwesentlich
  - Weiterhin O(n) Zusatzspeicher im schlimmsten Fall

Weitere Sortierverfahren

- **Heapsort**
  - Variante von Sortieren durch **Direkte Auswahl**
  - Verwendet rekursive Datenstruktur: **Minimum-Heap**
  - Minimum aus Heap mit n Werten entfernen: O(log n)
  - Heapaufbau bottom-up O(n), Teilheaps gleich groß
  - Sortieren: n mal Minimum aus Heap nehmen
  - Nicht sortierte Bereich wird für Heap verwendet
  - O(n log n) Zeitaufwand im schlimmsten Fall, O(1) Zusatzspeicher
  - Etwa zwei mal so langsam wie Mergesort



- **Shellsort**

  - Verbesserung von **Direktem Einfügen**, nähert sich schrittweise bestem Fall an
  - Direktes Einfügen wird mehrfach auf immer größer werdenden Teilfolgen von a angewendet, z.B. n/2, n/4, …, n/n = 1 (Beispiel: n = 16, 16 / 4 Teilfolgen)
  - Zeitaufwand hängt von Wahl der **Schrittweiten der Teilfolgen** ab
  - Teilfolge existiert für $O(n^{1,25})$, unbekannt ob Teilfolge für O(n log n) existiert
  - Shellsort ist einfach zu implementieren und recht gut für n < 1000
- **Bubblesort**:

  - Minimum von rechts aufsammeln und nach links durchrutschen lassen
  - schlechte Implementierungsvariante von **Direkter Auswahl**, kein Vorteil

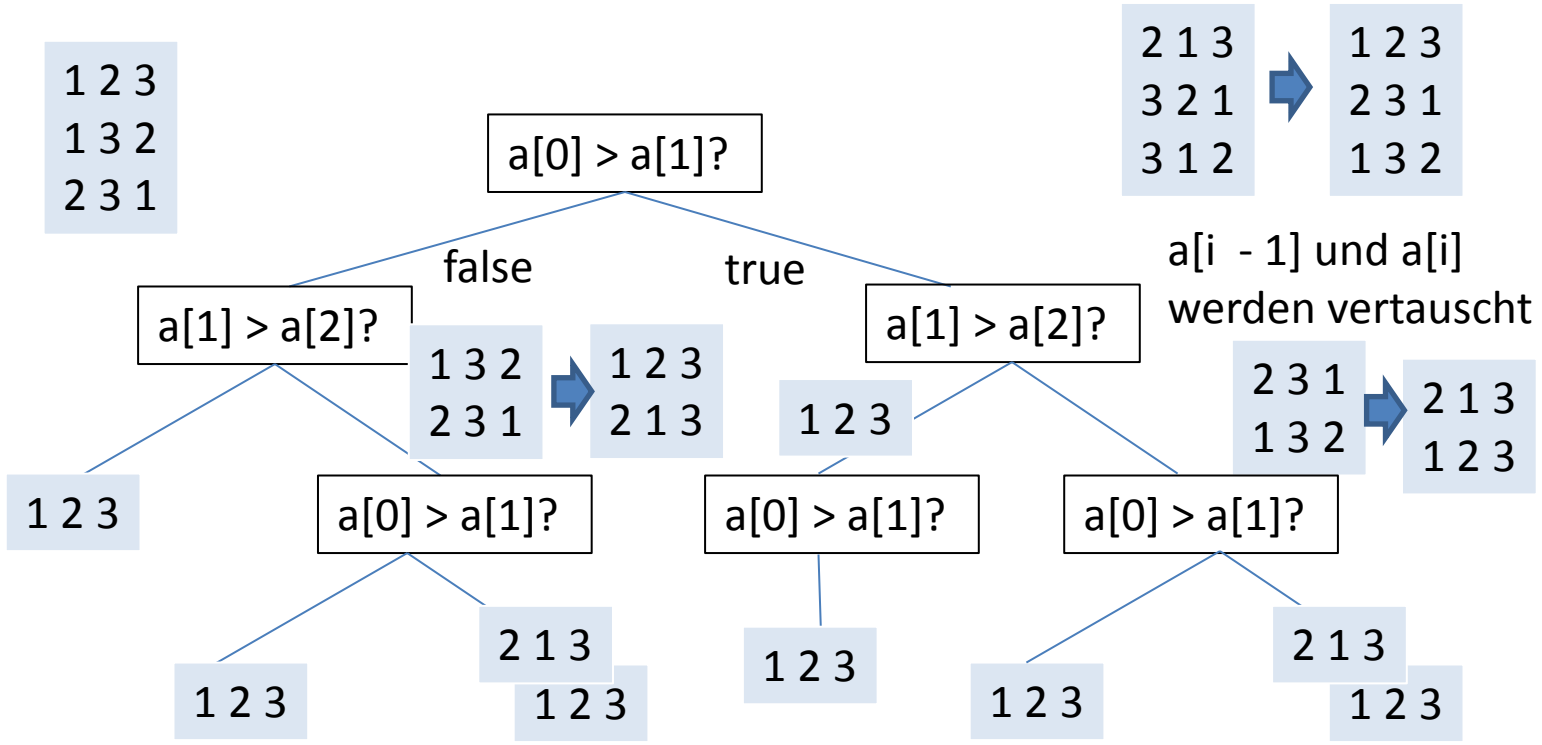
Sortieren

- Untere Schranke Sortieren: O(n log n) (genauer: Ω(n log n))
- Gilt für alle Algorithmen, die
  - auf Vergleiche von Werten der zu sortierenden Folge beruht
  - Werte der Folge sind Permutationen von {1, 2, .. , n – 1, n}
- Mergesort und Heapsort sind optimal hinsichtlich des Zeitaufwands.
- Heapsort ist zusätzlich optimal hinsichtlich des Speicheraufwands.
- Andere Verfahren:
  - Radix- und Bucketsort für ganze Zahlen mit konstanter Codierungsgrösse, z.B. 32-Bit, Sortieren mit O(n)
  - Da n beliebig groß ist, müssen beim Sortieren von Werten zwischen 1 bis n mindestens $\log_2 n$ Bits zur Codierung ganzer Zahlen verwendet werden. Vergleich zweier Werte kostet O(log n) Zeit. Insgesamt auch O(n log n) Zeitaufwand.

Untere Schranke Sortieren (1/4)

- Behauptung:
  - Untere Schranke Sortierproblem ist **O(n log n)**
- Probleme:
  - Unendliche viele Sortieralgorithmen
  - Unendliche viele Eingaben der Größe (bei reellen Zahlen)
  - Unendliche viele Eingaben, da n beliebig groß wird
- Vereinfachungen
  - Betrachte Folgen mit n Werten bestehend aus Zahlen von 1 bis n
  - Keine doppelten Vorkommen von Werten
  - Zähle nur Vergleiche zwischen Werten der Folge
- Hilfsmittel: **Entscheidungsbaum**
  - Protokolliert für einen Algorithmus und alle Eingaben der Größe n alle durchgeführten Vergleiche

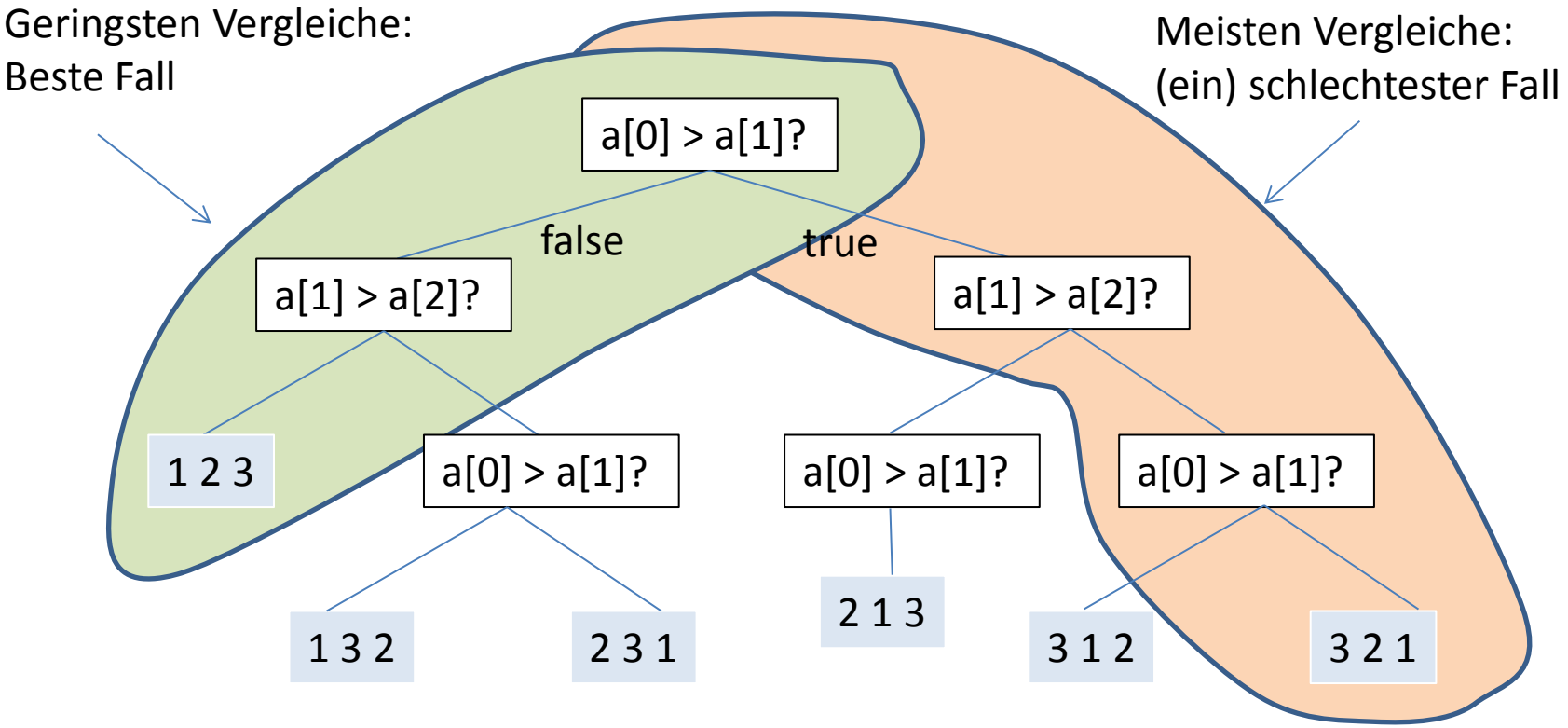
Untere Schranke Sortieren (2/4)



6 Pfade insgesamt. Jede der 3! Eingaben wird individuell erkannt und sortiert!

1 2 3 | 1 3 2 | 2 3 1 | 2 1 3 | 3 1 2 | 3 2 1

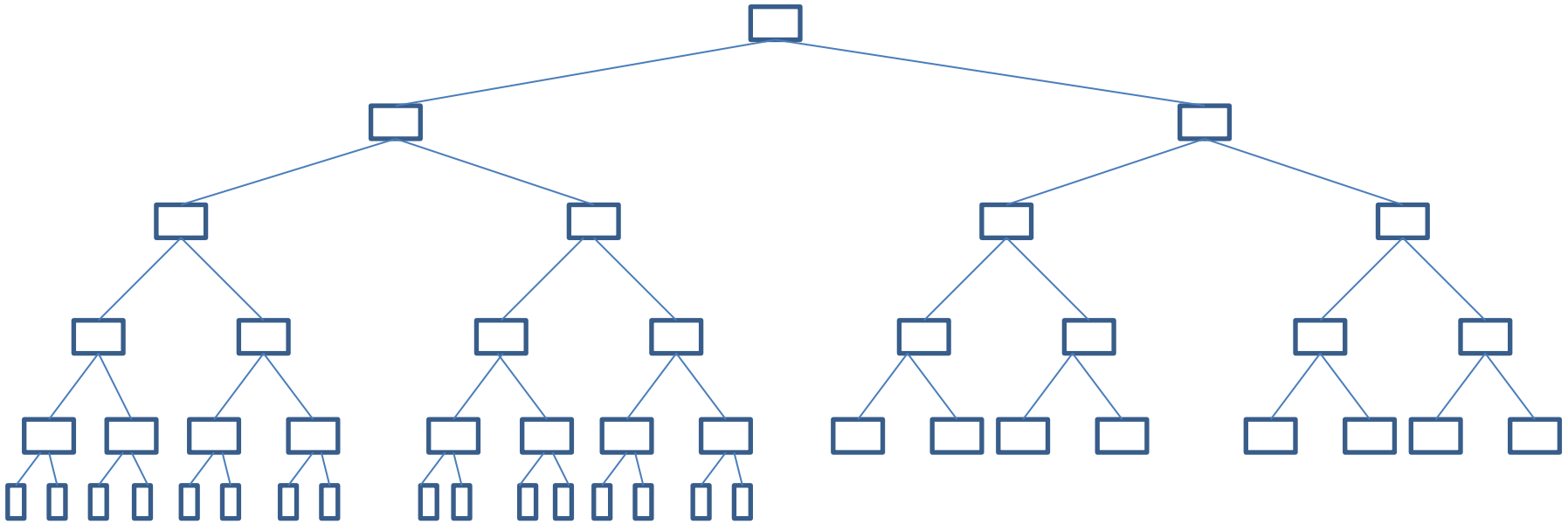
Untere Schranke Sortieren (3/4)



Wie muss ein Entscheidungsbaum aussehen, damit der längste Pfad (schlechteste Fall) möglichst wenig Vergleiche enthält?

Untere Schranke Sortieren (4/4)

Antwort: Entscheidungsbaum muss möglichst in die Breite wachsen.
Lediglich unterste Ebene kann unvollständig sein. Beispiel n = 4, n! = 24 Pfade



Anzahl Pfade verdoppelt sich mit jeder Ebene. Insgesamt n! verschiedene Pfade.
Anzahl Ebenen: $\log_2(n!) + 1$
Im Beispiel: $\log_2(4!) + 1 \leq \log_2(32) + 1 = 5 + 1 = 6$
Jeder Sortieralgorithmus hat im schlimmsten Fall mindestens $O(\log_2 n!)$ Schritte
Die **untere Schranke** des Sortierproblems ist $\boldsymbol{O(log_2 n!)}$. (? = $O(n \log n)$)

Rekurrenzgleichungen

Rekurrenz	Lösung in O-Notation
$T(n) = T(n-1) + c$	$\Theta(n^2)$
$T(n) = T\left(\frac{n}{2}\right) + c$	$\Theta(\log n)$
$T(n) = T\left(\frac{n}{2}\right) + nc$	$\Theta(n)$
$T(n) = 2T\left(\frac{n}{2}\right) + nc$	$\Theta(n \log n)$
$T(n) = T(an) + T(bn) + nc, a + b < 1$	$\Theta(n)$
$T(n) = 2T(n-1) + c$	$\Theta(2^n)$