# Sortieralgorithmen

Einführung in die Programmierung 1

Wintersemester 21/22

# Überblick

- Elementare Sortieralgorithmen
- Effiziente Sortieralgorithmen

# Elementare Sortieralgorithmen

# Annahmen

- Allgemein
  - Daten (sog. Schlüssel) werden in einem Array gespeichert
  - Auf den Daten ist eine Ordnungsrelation „≤“ definiert
- Daten
  - Alle Beispiele zuerst mit ganzen Zahlen
  - Die Algorithmen funktionieren auch auf anderen Datentypen
    - Z. B. mit kleinen Änderungen bei Vergleichen
- Implementierung in Methoden
  - Es werden immer existierende Arrays übergeben
    - Es wird daher nicht explizit auf `null` überprüft
  - O. B. d. A. wird immer aufsteigend sortiert

# Bubblesort – Idee

- Durchlauf
  - Array von links nach rechts durchlaufen
  - Es werden immer **zwei benachbarte** Elemente betrachtet
  - Wenn sie in falscher Reihenfolge vorkommen, werden sie vertauscht
- Der obige Schritt wird so oft wiederholt, bis das Array komplett sortiert ist
  - Das letzte Element des vorherigen Durchlaufs muss nicht mehr beachtet werden (ist an der richtigen Position)

# Bubblesort – Implementierung

```
private static void bubbleSort(int[] data) {
    for (int i = 0; i < data.length - 1; i++) {
        for (int j = 0; j < data.length - i - 1; j++) {
            if (data[j] > data[j + 1]) {
                exchange(data, j, j + 1);
            }
        }
    }
}

private static void exchange(int[] a, int i, int j) {
    int swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

# Bubblesort – Beispiel (1)

- Ursprüngliches Array [73, 2, 19, 96, 11, 5, 73, 66, 99, 1, 72, 30]
- Innerste Schleife (für i = 0) visualisiert

```
[2, 73, 19, 96, 11, 5, 73, 66, 99, 1, 72, 30]
[2, 19, 73, 96, 11, 5, 73, 66, 99, 1, 72, 30]
[2, 19, 73, 11, 96, 5, 73, 66, 99, 1, 72, 30]
[2, 19, 73, 11, 5, 96, 73, 66, 99, 1, 72, 30]
[2, 19, 73, 11, 5, 73, 96, 66, 99, 1, 72, 30]
[2, 19, 73, 11, 5, 73, 66, 96, 99, 1, 72, 30]
[2, 19, 73, 11, 5, 73, 66, 96, 99, 1, 72, 30]
[2, 19, 73, 11, 5, 73, 66, 96, 1, 99, 72, 30]
[2, 19, 73, 11, 5, 73, 66, 96, 1, 72, 99, 30]
[2, 19, 73, 11, 5, 73, 66, 96, 1, 72, 30, 99]
```

```
private static void bubbleSort(int[] data) {
    for (int i = 0; i < data.length - 1; i++) {
        for (int j = 0; j < data.length - i - 1; j++) {
            if (data[j] > data[j + 1]) {
                exchange(data, j, j + 1);
            }
        }
    }
}
```

Nach einem kompletten Durchlauf steht das Maximum (99) ganz rechts

# Bubblesort – Beispiel (2)

- Ursprüngliches Array

```
[73  2  19  96  11  5  73  66  99  1  72  30]
```

- Array jeweils nach innerer for-Schleife ausgegeben

```
[2  19  73  11  5  73  66  96  1  72  30  99]
[2  19  11  5  73  66  73  1  72  30  96  99]
[2  11  5  19  66  73  1  72  30  73  96  99]
[2  5  11  19  66  1  72  30  73  73  96  99]
[2  5  11  19  1  66  30  72  73  73  96  99]
[2  5  11  1  19  30  66  72  73  73  96  99]
[2  5  1  11  19  30  66  72  73  73  96  99]
[2  1  5  11  19  30  66  72  73  73  96  99]
[1  2  5  11  19  30  66  72  73  73  96  99]
[1  2  5  11  19  30  66  72  73  73  96  99]
[1  2  5  11  19  30  66  72  73  73  96  99]
```

# Bubblesort – Analyse

- Zwei verschachtelte Schleifen
  - Werden immer komplett durchlaufen
    - Unabhängig von der Eingabesequenz
  - Laufzeit ist in **$O(n^2)$**
    - Anweisungen in der innersten for-Schleife haben konstante Laufzeit

# Bubblesort – Optimierung

```
private static void bubbleSort2(int[] data) {
    boolean notSorted = true;
    while (notSorted) {
        notSorted = false;
        for (int i = 0; i < data.length - 1; i++) {
            if (data[i] > data[i + 1]) {
                exchange(data, i, i + 1);
                notSorted = true;
            }
        }
    }
}
```

- Verbesserung
  - Abbruch, wenn in einem Durchlauf keine Elemente mehr vertauscht werden

# Bubblesort optimiert – Beispiel

- Ursprüngliches Array

```
[73  2  19  96  11  5  73  66  99  1  72  30]
```

- Array jeweils nach der `for`-Schleife

```
[2  19  73  11  5  73  66  96  1  72  30  99]
[2  19  11  5  73  66  73  1  72  30  96  99]
[2  11  5  19  66  73  1  72  30  73  96  99]
[2  5  11  19  66  1  72  30  73  73  96  99]
[2  5  11  19  1  66  30  72  73  73  96  99]
[2  5  11  1  19  30  66  72  73  73  96  99]
[2  5  1  11  19  30  66  72  73  73  96  99]
[2  1  5  11  19  30  66  72  73  73  96  99]
[1  2  5  11  19  30  66  72  73  73  96  99]
[1  2  5  11  19  30  66  72  73  73  96  99]
```

# Bubblesort optimiert – Analyse

- Schlechtester Fall (absteigend sortiertes Array)
  - Beide Schleifen müssen immer durchlaufen werden
  - Laufzeit ist in **$O(n^2)$**
- Durchschnittlicher Fall
  - Ähnlich schlechtestem Fall
- Bester Fall (aufsteigend sortiertes Array)
  - Optimierte Variante durchläuft Array genau einmal
  - Laufzeit ist in **$O(n)$**

# Bester Fall – Vergleich

- Bubblesort
  - Ursprüngliches Array

    ```
    [1  2  3  4  5  6  7  8  9  10]
    ```

  - Ausgabe nach innerer Schleife

    ```
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    [1  2  3  4  5  6  7  8  9  10]
    ```

- Bubblesort optimiert
  - Ursprüngliches Array

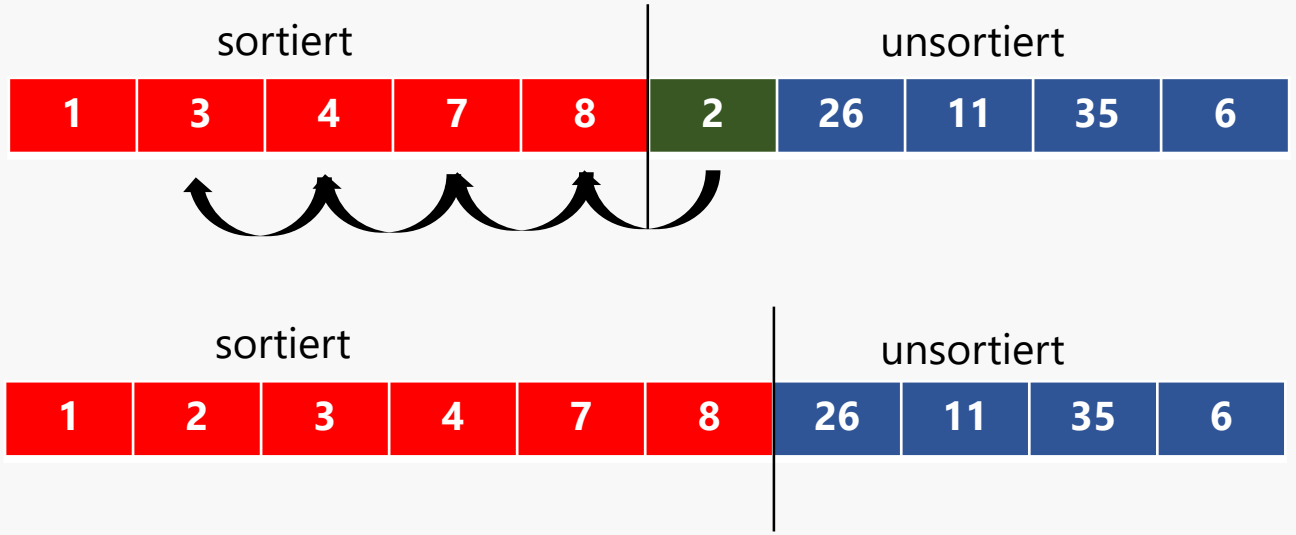
    ```
    [1  2  3  4  5  6  7  8  9  10]
    ```

  - Ausgabe nach innerer Schleife

    ```
    [1  2  3  4  5  6  7  8  9  10]
    ```

# Insertionsort – Idee

- Insertionsort nimmt aus dem unsortierten Teil der Eingabesequenz das erste Element
- Fügt das Element an der richtigen Stelle in die bisher sortierte Sequenz ein



# Insertionsort – Implementierung

```
private static void insertionSort(int[] data) {
    for (int i = 1; i < data.length; i++) {
        for (int j = i; j > 0 && data[j] < data[j - 1]; j--) {
            exchange(data, j, j - 1);
        }
    }
}
```

# Insertionsort – Beispiel (1)

```
private static void insertionSort(int[] data) {
    for (int i = 1; i < data.length; i++) {
        for (int j = i; j > 0 && data[j] < data[j - 1]; j--) {
            exchange(data, j, j - 1);
        }
    }
}
```

- Ursprüngliches Array

  [73, 2, 19, 96, 11, 5, 73, 66, 99, 1, 72, 30]
- Innerste Schleife (für i = 1) visualisiert

  [**2, 73**, 19, 96, 11, 5, 73, 66, 99, 1, 72, 30]
- Innerste Schleife (für i = 2) visualisiert

  [2, **19, 73**, 96, 11, 5, 73, 66, 99, 1, 72, 30]
- Innerste Schleife (für i = 4) visualisiert

  [2, 19, 73, **11, 96**, 5, 73, 66, 99, 1, 72, 30]

  [2, 19, **11, 73**, 96, 5, 73, 66, 99, 1, 72, 30]

  [2, **11, 19**, 73, 96, 5, 73, 66, 99, 1, 72, 30]

# Insertionsort – Beispiel (2)

- Ursprüngliches Array

```
[73  2  19  96  11  5  73  66  99  1  72  30]
```

- Array jeweils nach der inneren `for`-Schleife

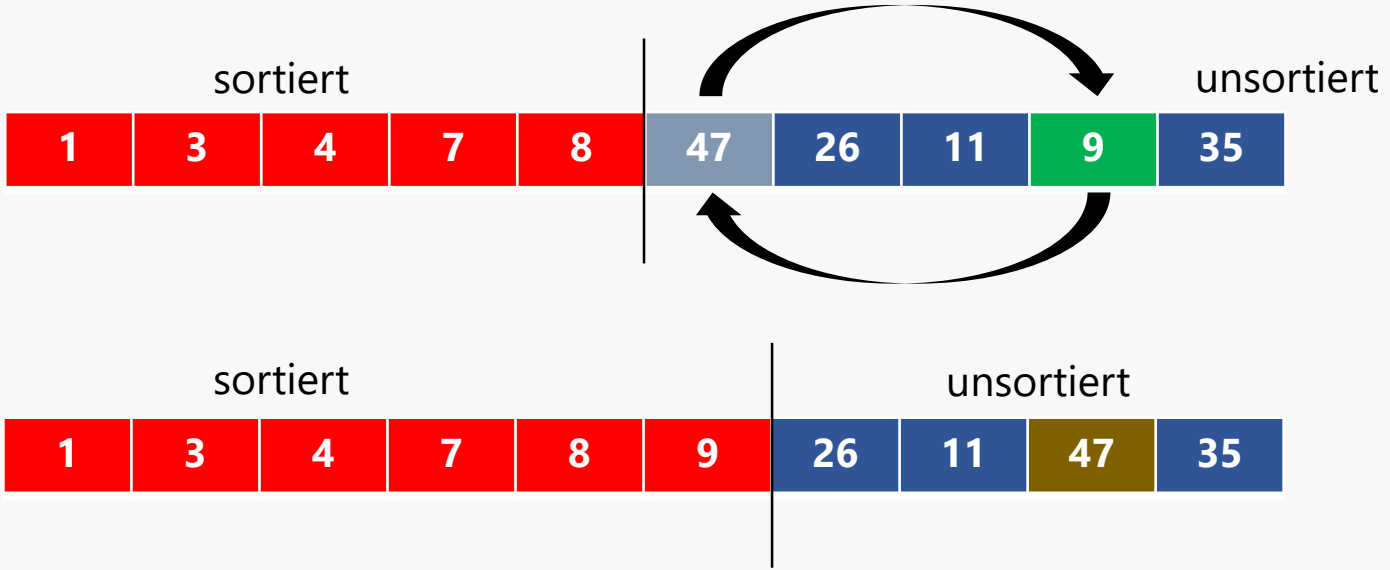
```
[2  73  19  96  11  5  73  66  99  1  72  30]
[2  19  73  96  11  5  73  66  99  1  72  30]
[2  19  73  96  11  5  73  66  99  1  72  30]
[2  11  19  73  96  5  73  66  99  1  72  30]
[2  5  11  19  73  96  73  66  99  1  72  30]
[2  5  11  19  73  73  96  66  99  1  72  30]
[2  5  11  19  66  73  73  96  99  1  72  30]
[2  5  11  19  66  73  73  96  99  1  72  30]
[1  2  5  11  19  66  73  73  96  99  72  30]
[1  2  5  11  19  66  72  73  73  96  99  30]
[1  2  5  11  19  30  66  72  73  73  96  99]
```

# Insertionsort – Analyse

- Schlechtester Fall (absteigend sortiertes Array übergeben)
  - Schleifen werden komplett durchlaufen
  - Laufzeit ist in **$O(n^2)$**
- Durchschnittlicher Fall
  - Wie schlechtester Fall
- Bester Fall (aufsteigend sortiertes Array übergeben)
  - Innere Schleife wird nie betreten (Bedingung wertet immer gleich auf false aus)
  - Laufzeit ist in **$O(n)$**

# Selectionsort – Idee

- Selectionsort wählt immer das kleinste Element aus dem noch nicht sortierten Teil der Eingabesequenz
- Tauscht das Element an der ersten Position nach dem sortierten Teil mit dem kleinsten Element



# Selectionsort – Implementierung

```
private static void selectionSort(int[] data) {
    for (int i = 0; i < data.length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < data.length; j++) {
            if (data[j] < data[min]) {
                min = j;
            }
        }
        exchange(data, i, min);
    }
}
```

# Selectionsort – Beispiel (1)

- Ursprüngliches Array

  [73, 2, 19, 96, 11, 5, 73, 66, 99, 1, 72, 30]
- Innerste Schleife (für i = 0) visualisiert
  - Index des Minimums = 9
  - Array nach dem Tausch: [**1**, 2, 19, 96, 11, 5, 73, 66, 99, **73**, 72, 30]
- Innerste Schleife (für i = 1) visualisiert
  - Index des Minimums = 1
  - Array nach dem Tausch: [1, **2**, 19, 96, 11, 5, 73, 66, 99, 73, 72, 30]
- Innerste Schleife (für i = 2) visualisiert
  - Index des Minimums = 5
  - Array nach dem Tausch: [1, 2, **5**, 96, 11, **19**, 73, 66, 99, 73, 72, 30]

# Selectionsort – Beispiel (2)

- Ursprüngliches Array

```
[73  2  19  96  11  5  73  66  99  1  72  30]
```

- Array jeweils nach der inneren `for`-Schleife

```
[1  2  19  96  11  5  73  66  99  73  72  30]
[1  2  19  96  11  5  73  66  99  73  72  30]
[1  2  5  96  11  19  73  66  99  73  72  30]
[1  2  5  11  96  19  73  66  99  73  72  30]
[1  2  5  11  19  96  73  66  99  73  72  30]
[1  2  5  11  19  30  73  66  99  73  72  96]
[1  2  5  11  19  30  66  73  99  73  72  96]
[1  2  5  11  19  30  66  72  99  73  73  96]
[1  2  5  11  19  30  66  72  73  99  73  96]
[1  2  5  11  19  30  66  72  73  73  99  96]
[1  2  5  11  19  30  66  72  73  73  96  99]
```

# Selectionsort – Analyse

- Zwei verschachtelte Schleifen
  - Werden immer komplett durchlaufen
    - Unabhängig von der Eingabesequenz
  - Laufzeit ist in **$O(n^2)$**
  - Bester Fall, durchschnittlicher Fall und schlechtester Fall daher gleich!
- Anzahl der Aufrufe von exchange
  - Linear, unabhängig von der Eingabesequenz
  - Bei Insertionsort z. B. abhängig von der Eingabesequenz
    - Kann dort auch quadratisch sein

# Elementare Sortierverfahren – Vergleich

- Bubblesort
  - Einfach zu implementieren
- Insertionsort
  - Lineare Laufzeit bei vorsortierten Sequenzen
- Selectionsort
  - Anzahl der Vertauschungen steigt linear an
  - Besser (im Durchschnitt) als Insertionsort bzw. Bubblesort, falls Vertauschungen sehr aufwändig

# Stabilität von Sortieralgorithmen (1)

- Stabiles Sortierverfahren
  - Die Reihenfolge der Datensätze (komplexe Datentypen), deren Sortierschlüssel gleich sind, bleibt bewahrt
- Beispiel für komplexeren Datentyp
  - Personaldaten
    - Name
    - Abteilungsnummer
  - Beide Informationen werden gemeinsam gespeichert
    - Implementierungsmöglichkeit wird bei der Objektorientierung noch besprochen

# Stabilität von Sortieralgorithmen (2)

- Beispiel (Abteilungsnummer + Name)
  - Liste von Personaldaten mit Abteilungsnummer
    - Innerhalb der Abteilung sortiert nach Vornamen

| 3 | 4 | 2 | 3 | 1 | 3 |
|---|---|---|---|---|---|
| **Daniel** | **Maria** | **Paul** | **Erika** | **Anton** | **Sarah** |

- Sortierung nach Abteilungsnummer mit stabilem Sortierverfahren
  - Sortierung innerhalb einer Abteilung sollte erhalten bleiben

| 1 | 2 | 3 | 3 | 3 | 4 |
|---|---|---|---|---|---|
| **Anton** | **Paul** | **Daniel** | **Erika** | **Sarah** | **Maria** |

# Stabilität – Beispiele

- Sequenz vereinfacht **3 4 2 3 1 3**
- Vergleich der bisherigen Algorithmen
  - Ausgaben nach inneren Schleifen

| Bubblesort | Insertionsort | Selectionsort |
|---|---|---|
| **3 4 2 3 1 3** | **3 4 2 3 1 3** | **3 4 2 3 1 3** |
| **3 2 3 1 3 4** | **3 4 2 3 1 3** | **1 4 2 3 3 3** |
| **2 3 1 3 3 4** | **2 3 4 3 1 3** | **1 2 4 3 3 3** |
| **2 1 3 3 3 4** | **2 3 3 4 1 3** | **1 2 3 4 3 3** |
| **1 2 3 3 3 4** | **1 2 3 3 4 3** | **1 2 3 3 4 3** |
| **1 2 3 3 3 4** | **1 2 3 3 3 4** | **1 2 3 3 3 4** |
| stabil | stabil | **nicht stabil** |

# Elementare Sortierverfahren – Laufzeiten

- Beispiele für Laufzeiten
  - Arrays der Länge *n* mit Zufallszahlen im Bereich 0 .. *n*
  - Laufzeiten für einen Algorithmus
    - Mittelwert von zehn Messungen pro Eingabegröße
  - System - Intel Core i7 9750H, Windows 10, Java 14

| Eingabegröße n | Bubblesort (sec) | Insertionsort (sec) | Selectionsort (sec) |
|---|---|---|---|
| 10000 | 0,10 | 0,01 | 0,03 |
| 20000 | 0,43 | 0,05 | 0,09 |
| 40000 | 1,80 | 0,19 | 0,36 |
| 80000 | 7,28 | 0,79 | 1,42 |
| 160000 | 29,17 | 3,14 | 5,66 |

# Mergesort

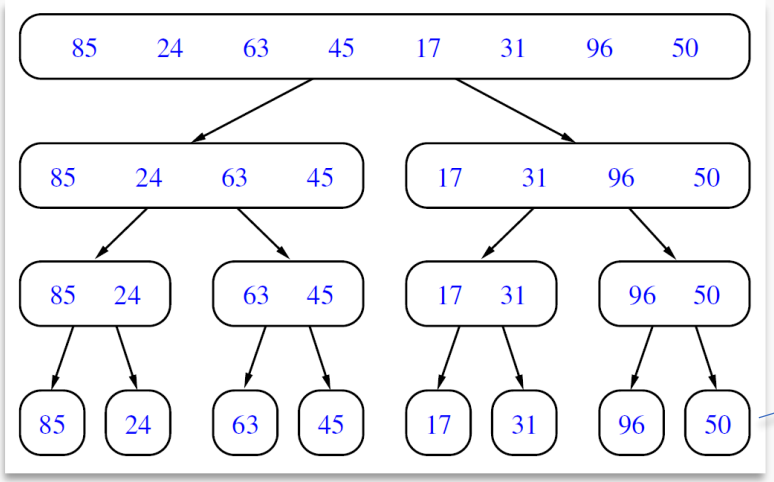
# Teile-und-herrsche-Verfahren

- Allgemeines Prinzip für Algorithmenentwurf
  - Teile ein Problem in Teilprobleme auf
  - Löse jedes Teilproblem unabhängig für sich
  - Kombiniere die Teillösungen zu einer Gesamtlösung für das ursprüngliche Problem
- Im Englischen unter *Divide-and-Conquer* bekannt
- Wenn ein Teile-und-herrsche-Verfahren für ein Problem bekannt ist
  - Lässt es sich schnell rekursiv implementieren
  - Basisfall ist erreicht, wenn das Teilproblem so einfach ist, dass die Lösung sofort angegeben werden kann

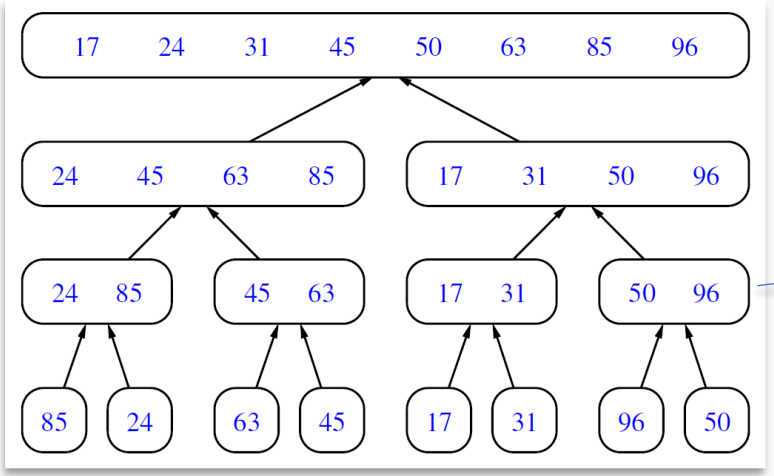
# Mergesort – Idee

- Ein Teile-und-herrsche-Verfahren
- Vorgehen
  - Teile ursprüngliche Menge an Daten in zwei Hälften
  - Sortiere jede Hälfte für sich (rekursiv)
  - Verschmelze die beiden sortierten Hälften
    - Vergleiche immer wieder die aktuell vordersten Elemente der beiden sortierten Hälften
    - Entnimm (bzw. verarbeite) das kleinere der beiden
    - Am Ende ergibt sich so eine geordnete Sequenz aus den zwei geordneten Hälften
- Wird auch als Top-Down-Mergesort bezeichnet

# Visuelles Beispiel für Ablauf (Überblick)

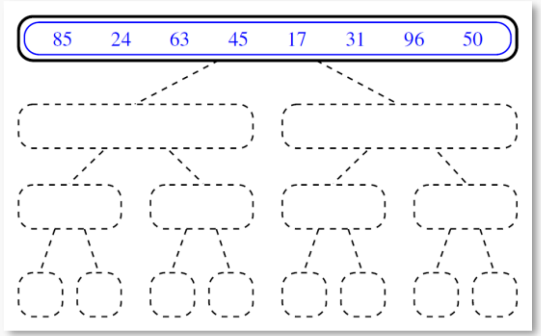


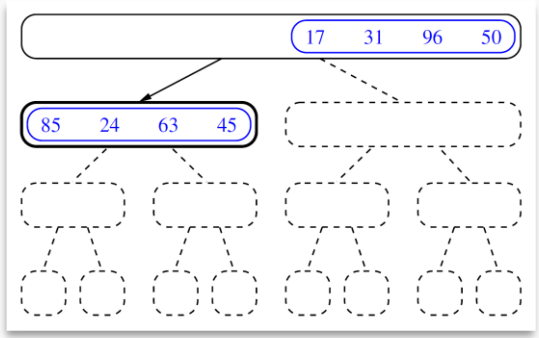

Solange aufteilen, bis nur mehr ein Element betrachtet wird

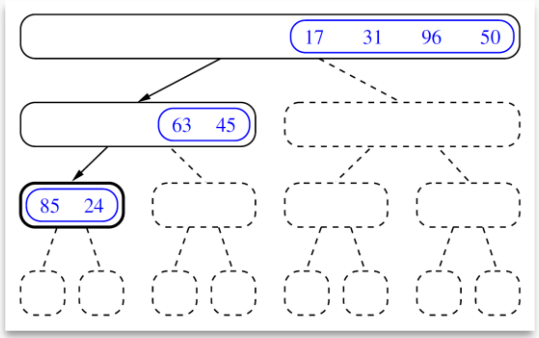


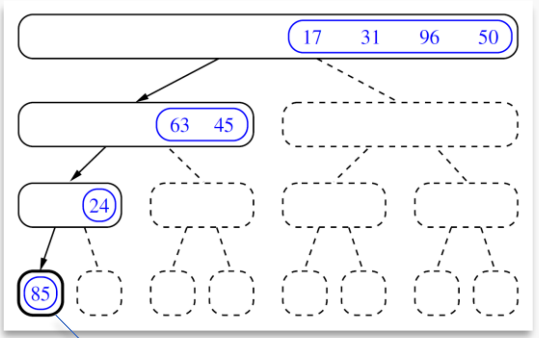

Verschmelzen von jeweils zwei sortierten Bereichen zu einer neuen sortierten Bereich

# Beispiel – Rekursive Aufrufe (1)

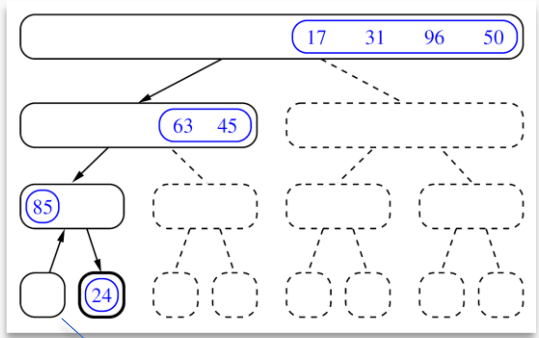


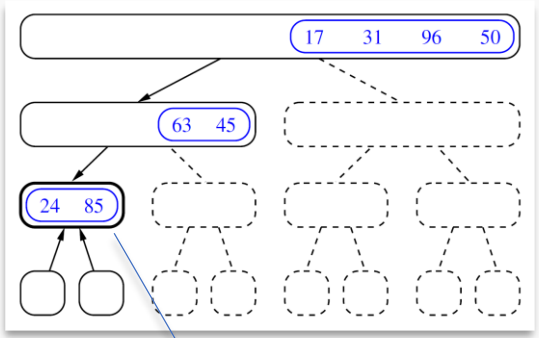




Aktueller Aufruf

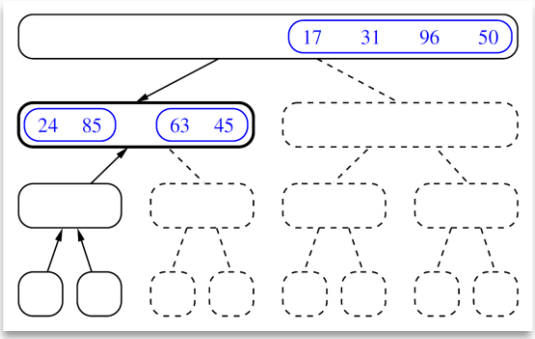


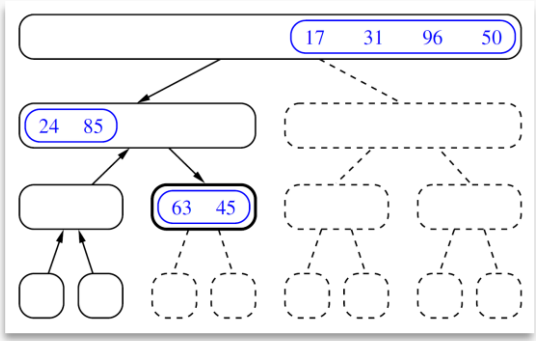

Beendeter Aufruf

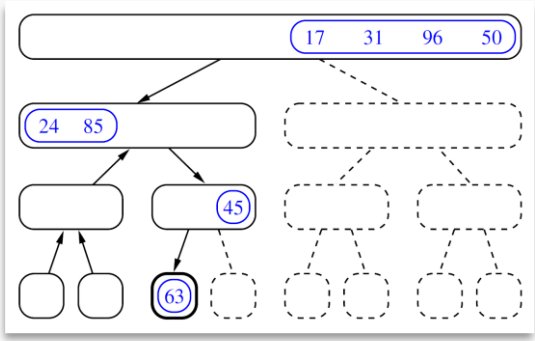


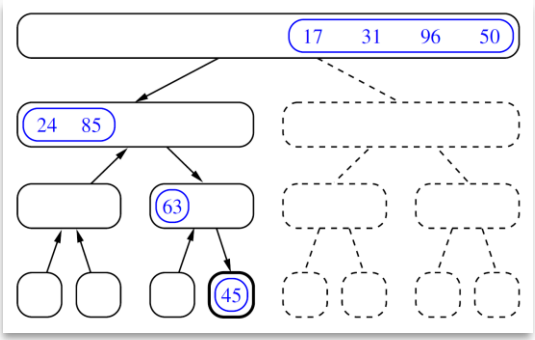

Nach Verschmelzen

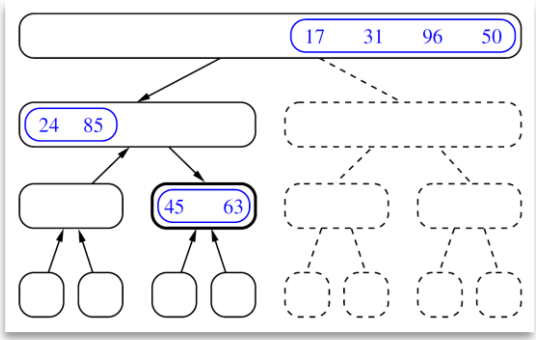
# Beispiel – Rekursive Aufrufe (2)

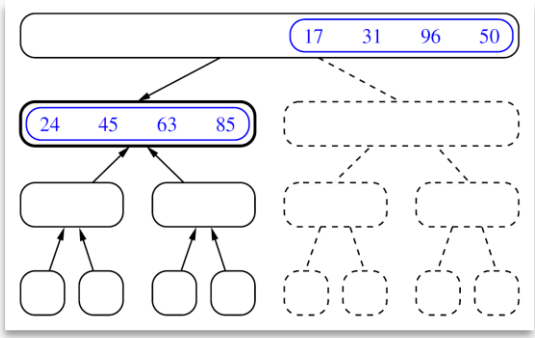


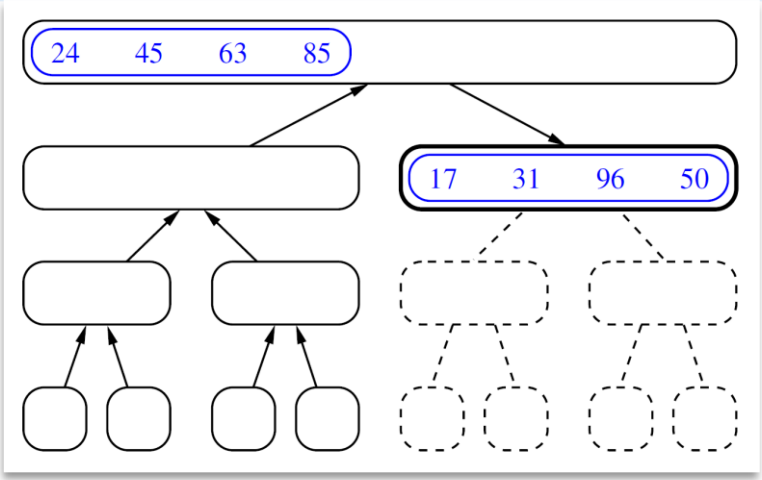



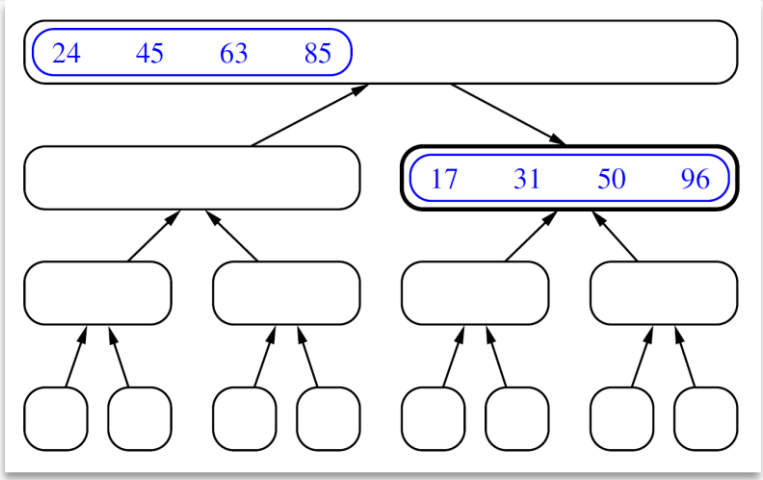



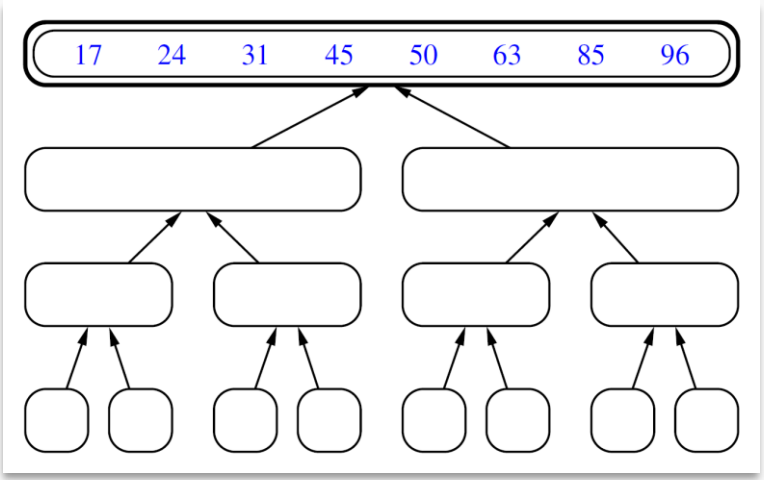



# Beispiel – Rekursive Aufrufe (3)



# Mergesort – Implementierung

```
private static void mergeSort(int[] data) {
    int[] help = new int[data.length];
    mergeSort(data, help, 0, data.length - 1);
}

private static void mergeSort(int[] data, int[] help, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    mergeSort(data, help, lo, mid);
    mergeSort(data, help, mid + 1, hi);
    merge(data, help, lo, mid, hi);
}

private static void merge(int[] data, int[] help, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++) {
        help[k] = data[k];
    }
    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) data[k] = help[j++];
        else if (j > hi) data[k] = help[i++];
        else if (help[j] < help[i]) data[k] = help[j++];
        else data[k] = help[i++];
    }
}
```

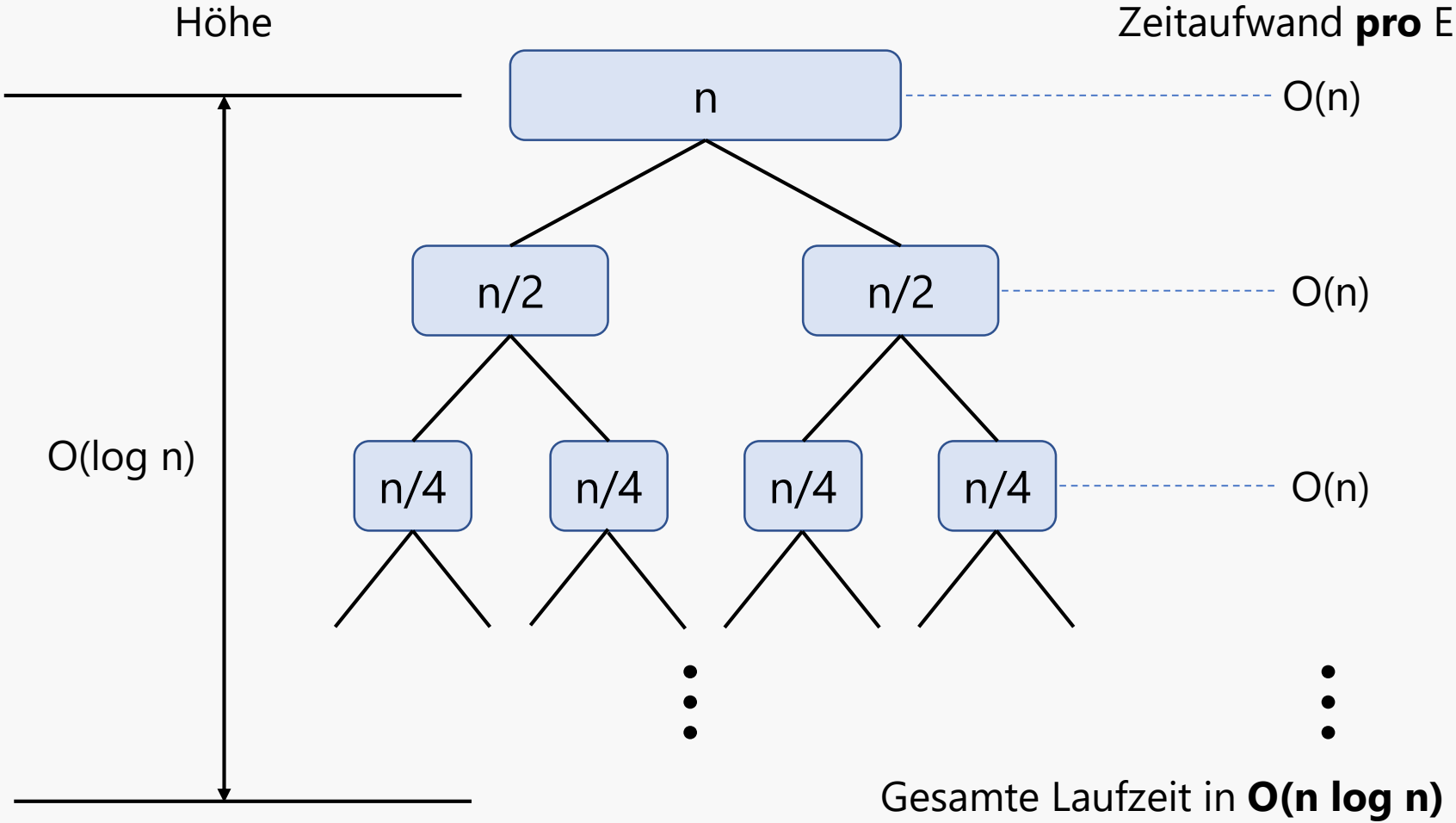
Aufruf mit einem Parameter (wie bisher)

Mergesort
Zuerst linke Hälfte sortieren,
dann rechte Hälfte sortieren,
zuletzt **sortierte** Hälften verschmelzen

merge (Verschmelzen zweier sortierter Bereiche)

# merge – Beispiel

# Mergesort – Analyse (1)



# Mergesort – Analyse (2)

- Laufzeit ist immer in O(n log n)
- Z. B. vorsortiertes Array bringt keinen Vorteil
  - Array wird immer geteilt
  - Verschmelzen wird immer durchgeführt

# Mergesort – Analyse (3)

- Laufzeit ist in **O(n log n)**
  - Aber mehr Speicherplatzbedarf durch Hilfsarray beim Verschmelzen, dieser liegt in O(n)
- *n log n* kann nicht unterboten werden, wenn wir das **allgemeine** Sortierproblem betrachten
  - Es müssen *n* Werte sortiert werden
  - Keine zusätzliche Information über den Wertebereich
- Mergesort ist ein stabiles Sortierverfahren

# Mergesort – alternative Variante

```
private static void mergeSortBU(int[] data) {
    int[] help = new int[data.length];
    for (int len = 1; len < data.length; len *= 2) {
        for (int lo = 0; lo < data.length - len; lo += len + len) {
            int mid = lo + len - 1;
            int hi = Math.min(lo + len + len - 1, data.length - 1);
            merge(data, help, lo, mid, hi);
        }
    }
}
```

- Es werden nur merge-Operationen durchgeführt
  - Zuerst 1er-Blöcke verschmelzen, dann 2er-Blöcke verschmelzen, dann 4er-Blöcke verschmelzen usw.
- Diese Variante wird als Bottom-Up-Mergesort bezeichnet

# Quicksort

# Quicksort – Idee

- Vorgehen (Teile-und-herrsche-Verfahren)
  - Aufteilen
    - Wähle ein Element *x* (Pivotelement) aus dem Array bzw. Abschnitt im Array
    - Teile den Abschnitt im Array in zwei Abschnitte $A_1$ und $A_2$, so dass gilt
      - $A_1$ enthält nur Elemente kleiner gleich x
      - $A_2$ enthält nur Elemente größer gleich x
  - Rekursive Aufrufe
    - Rekursiver Aufruf auf $A_1$ erzeugt sortierte Folge $B_1$
    - Rekursiver Aufruf auf $A_2$ erzeugt sortierte Folge $B_2$
    - Ergebnis am Ende im Array auf einer Rekursionsebene
      - $B_1$, x, $B_2$

# Quicksort – Implementierung

```
private static void quickSort(int[] data) {
    quickSort(data, 0, data.length - 1);
}

private static void quickSort(int[] data, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(data, lo, hi);
    quickSort(data, lo, j - 1);
    quickSort(data, j + 1, hi);
}

private static int partition(int[] data, int lo, int hi) {
    int k = lo, v = data[hi];
    for (int i = k; i < hi; i++) {
        if (data[i] < v) {
            exchange(data, i, k++);
        }
    }
    exchange(data, k, hi);
    return k;
}
```

Aufruf mit einem Parameter (wie bisher)

Quicksort
Zuerst partitionieren,
dann linke Hälfte sortieren,
dann rechte Hälfte sortieren

# Quicksort – Beispiel für Partitionierung

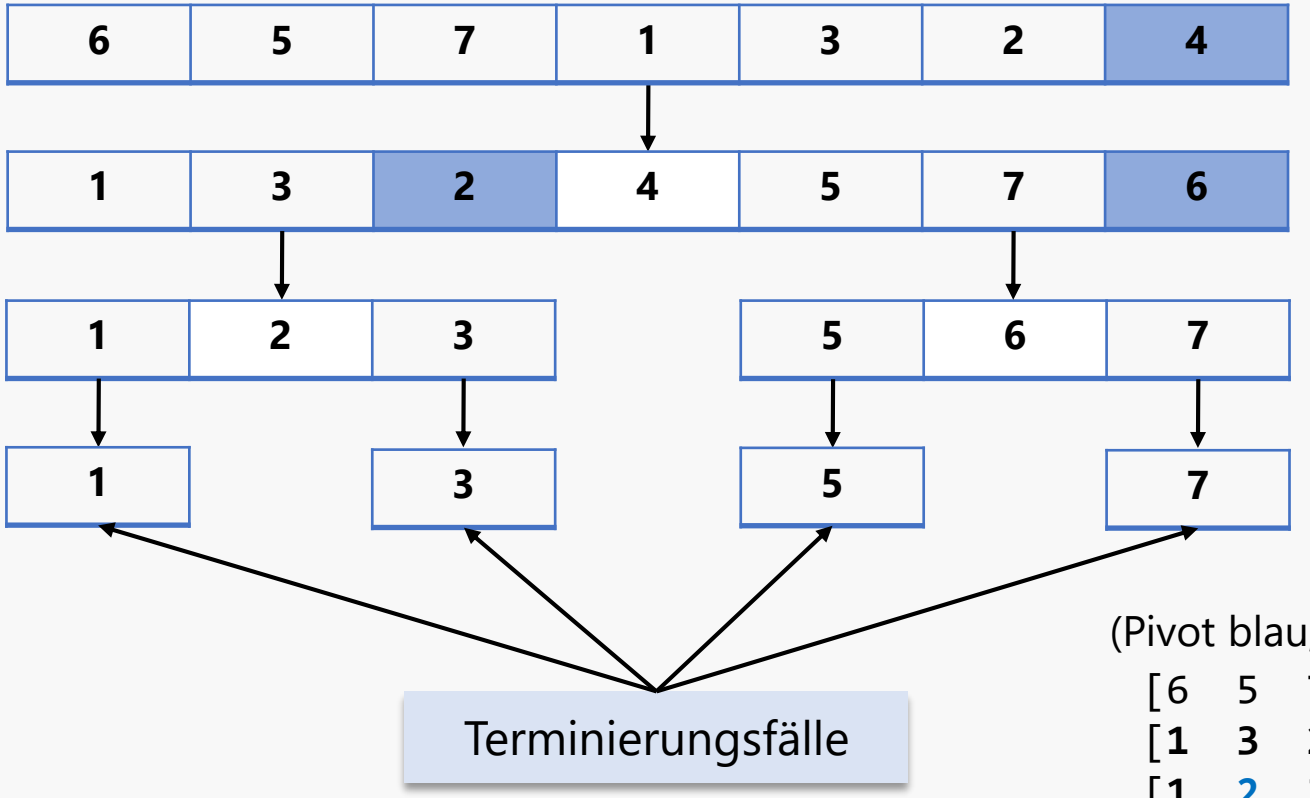

Pivotelement für Indexbereich 0 (lo) bis 6 (hi)

[6, 5, 7, 1, 3, 2, 4] i: 0 k: 0
[6, 5, 7, 1, 3, 2, 4] i: 1 k: 0
[6, 5, 7, 1, 3, 2, 4] i: 2 k: 0
[**1**, 5, 7, **6**, 3, 2, 4] i: 3 k: 1
[1, **3**, 7, 6, **5**, 2, 4] i: 4 k: 2
[1, 3, **2**, 6, 5, 7, **4**] i: 5 k: 3

```
private static int partition(int[] data, int lo, int hi) {
    int k = lo, v = data[hi];
    for (int i = k; i < hi; i++) {
        if (data[i] < v) {
            exchange(data, i, k++);
        }
    }
    exchange(data, k, hi);
    return k;
}
```

Letzter Aufruf von exchange tauscht Pivotelement mit dem Element an der Position k

# Quicksort – Beispiel für optimale Aufteilung




(Pivot blau, fixierte Werte rot)

[6 5 7 1 3 2 4]
[1 3 2 4 5 7 6]
[1 2 3 4 5 7 6]
[1 2 3 4 5 6 7]
[1 2 3 4 5 6 7]

# Quicksort – schlechte Aufteilung

- Ursprüngliches Array

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

- Ausgabe jeweils nach Partitionierung (Pivot blau hervorgehoben)

[1, 2, 3, 4, 5, 6, 7, 8, 9, **10**]
[1, 2, 3, 4, 5, 6, 7, 8, **9**, 10]
[1, 2, 3, 4, 5, 6, 7, **8**, 9, 10]
[1, 2, 3, 4, 5, 6, **7**, 8, 9, 10]
[1, 2, 3, 4, 5, **6**, 7, 8, 9, 10]
[1, 2, 3, 4, **5**, 6, 7, 8, 9, 10]
[1, 2, 3, **4**, 5, 6, 7, 8, 9, 10]
[1, 2, **3**, 4, 5, 6, 7, 8, 9, 10]
[1, **2**, 3, 4, 5, 6, 7, 8, 9, 10]

# Quicksort – Analyse (1)

- Schwierige Analyse
- Bester und durchschnittlicher Fall
  - **O(n log n)**
- Schlechtester Fall (vorherige Folie)
  - **$O(n^2)$**
  - Durch gute/geschickte Wahl des Pivotelements kann versucht werden, den schlechtesten Fall möglichst zu vermeiden, z. B.
    - Pivotelement wird aus dem Median von drei Elementen (linkes Element, mittleres Element, rechtes Element) errechnet
    - Zufälliges Pivotelement

# Quicksort – Analyse (2)

- Quicksort benötigt kein zusätzliches Hilfsarray
  - Alle Vertauschungen werden im Array (in-place) durchgeführt
- Quicksort ist nicht stabil
- Hinweis
  - Alle Aussagen beziehen sich auf die präsentierten Implementierungen!

# Vergleich Sortieralgorithmen (1)

- Arrays der Länge *n* mit Zufallszahlen im Bereich 0 … *n*
- Laufzeiten für einen Algorithmus
  - Mittelwert von zehn Messungen pro Eingabegröße
  - Extrapolierte Werte (rot), abgerundete Werte (blau)
- System - Intel Core i7 9750H, Windows 10, Java 14

# Vergleich Sortieralgorithmen (2)

| Eingabegröße n | Bubblesort (sec) | Insertionsort (sec) | Selectionsort (sec) | Mergesort (sec) | Quicksort (sec) |
|---|---|---|---|---|---|
| 10000 | 0,10 | 0,01 | 0,03 | 0,00 | 0,00 |
| 20000 | 0,43 | 0,05 | 0,09 | 0,00 | 0,00 |
| 40000 | 1,80 | 0,19 | 0,36 | 0,00 | 0,00 |
| 80000 | 7,28 | 0,79 | 1,42 | 0,00 | 0,00 |
| 160000 | 29,17 | 3,14 | 5,66 | 0,01 | 0,01 |
| 320000 | 117 | 13 | 23 | 0,03 | 0,02 |
| 640000 | 468 | 52 | 92 | 0,06 | 0,05 |
| 1280000 | 1872 | 208 | 368 | 0,13 | 0,10 |
| 2560000 | 7488 | 832 | 1472 | 0,26 | 0,20 |
| 5120000 | 29952 | 3328 | 5888 | 0,55 | 0,43 |
| 10240000 | 119808 | 13312 | 23552 | 1,13 | 0,89 |

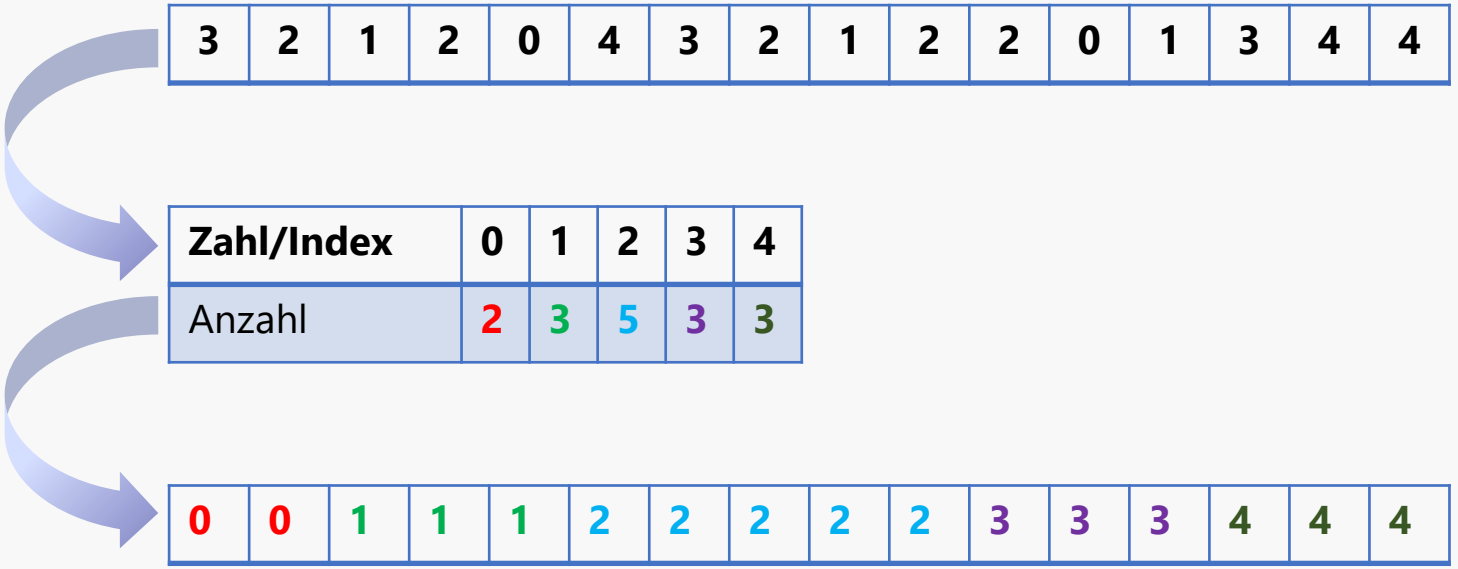
# Sortieren in linearer Zeit

# Schneller sortieren

- **O(n log n)** kann unterboten werden
- Aber
  - Zusätzliche Annahme treffen bzw. Information ausnutzen
- Annahmen
  - Wir gehen von *n* Zahlen im Bereich von 0 bis *max* aus
  - *max* << *n*
  - Zusätzliches Hilfsarray `counts` der Länge *max* + 1

# Countingsort – Idee

- Zähle in einem Hilfsarray mit, wie oft eine Zahl vorkommt
- Schreibe jede Zahl so oft in das ursprüngliche Array
  - Vereinfachte Version von Countingsort!
- Beispiel



# Countingsort – Implementierung

```
private static void countingSort(int[] data, int max) {
    int[] counts = new int[max + 1];
    for (int i = 0; i < data.length; i++) {
        counts[data[i]]++;
    }
    int i = 0;
    for (int j = 0; j <= max; j++) {
        for (int k = 0; k < counts[j]; k++) {
            data[i++] = j;
        }
    }
}
```

O)n)

O(n)

# Countingsort – Analyse

- Eine Schleife für das Befüllen des `counts`-Arrays
  - **O(n)**
- Zwei Schleifen für das Zurückschreiben
  - Es können aber nur *n* Zahlen geschrieben werden
  - **O(n)**
- Beide Teile werden hintereinander ausgeführt
  - **O(n)**
- Aber zusätzlicher Speicher notwendig
  - Vernachlässigbar, wenn $max << n$