

Algorithmen und Datenstrukturen

6. SORTIEREN

Sortieren

- Grundlegendes Problem in der Informatik
- Aufgabe:
 - ◆ Ordnen von Dateien mit Datensätzen, die Schlüssel enthalten
 - ◆ Umordnen der Datensätze so, dass klar definierte Ordnung der Schlüssel (numerisch/alphabetisch) besteht
- Vereinfachung: nur Betrachtung der Schlüssel, z.B. Feld von **int**-Werten

Sortieren - Grundbegriffe

- Verfahren
 - ◆ Intern: in Hauptspeicherstrukturen (Felder, Listen)
 - ◆ Extern: Datensätze auf externen Medien (Festplatte etc.)

- Annahmen
 - ◆ Totale Ordnung (Zahlen, alphabetisch, etc.)
 - ◆ Aufsteigend
 - ◆ Platzbedarf konstant für jedes Element

Sortieren - Problembeschreibung

Problem – Sortierproblem

Eingabe: Folge von Zahlen $\langle a_1, \dots, a_n \rangle$

Ausgabe: Permutation $\langle a_1', \dots, a_n' \rangle$ der Zahlen mit der

Eigenschaft: $a_1' \leq a_2' \leq \dots \leq a_n'$

- Sortierung nach einem Schlüssel (key), z.B. Zahlen
- Übertragbar (in Programmen) auf beliebige Datenstrukturen (mit Schlüssel)

Stabilität

Ein Sortierverfahren heißt stabil,
wenn es die relative Reihenfolge gleicher Schlüssel
in der Datei beibehält.

Beispiel: alphabetisch geordnete Liste von Personen soll nach Alter sortiert werden ! Personen mit gleichem Alter weiterhin alphabetisch geordnet:

Name	Alter		Name	Alter
Aristoteles	24		Aristoteles	24
Platon	28		Platon	28
Sokrates	30		Theophrastos	28
Theophrastos	28		Sokrates	30

Algorithmen und Datenstrukturen

6.1 ALGORITHMEN FÜR VERGLEICHSBASIERTES SORTIEREN

Vergleichsbasiertes Sortieren

- Wichtigster Spezialfall des Sortierproblems
- Zur Sortierung können nur **direkte** Vergleiche **zweier** Werte benutzt werden
- Der Wertebereich der Schlüssel kann beliebig sein

Problem: Vergleichsbasiertes Sortieren

Eingabe: Array ganzer Zahlen

Ausgabe: Sortiertes Array derselben Zahlen
(Mehrfachvorkommen bleiben erhalten)

Anmerkung: einige Sortierverfahren sind effizienter wenn Listen anstatt Arrays benutzt werden

Sortierinterface in Java

```
public interface Sort {  
  
    /**  
     * sorts the given array.  
     * @param f- array to sort.  
     */  
    public void execute(int[] f);  
}
```

Überblick Sortierverfahren

- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort

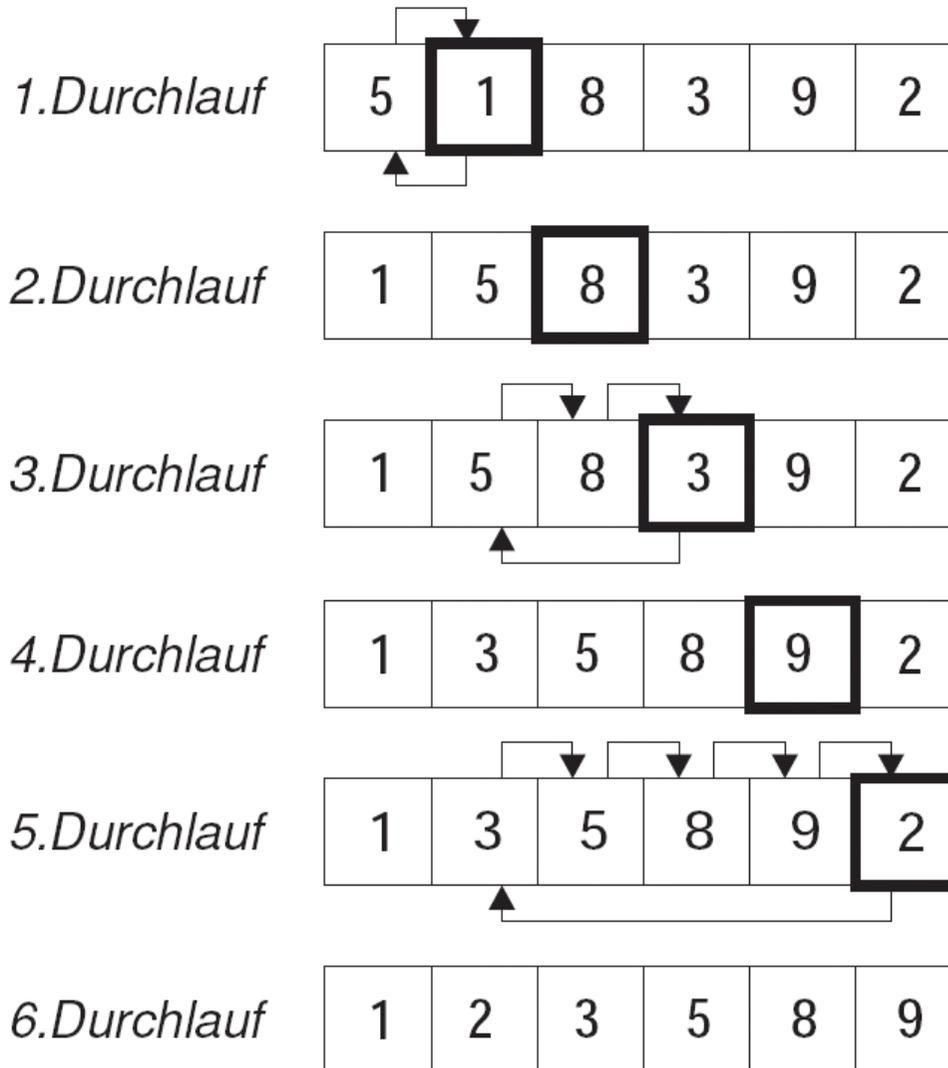
Algorithmen und Datenstrukturen

6.1.1 INSERTION SORT

Sortieren durch Einfügen: Das Prinzip

- Idee
 - ◆ Umsetzung der typischen menschlichen Vorgehensweise, etwa beim Sortieren eines Stapels von Karten:
 1. Starte mit der ersten Karte einen neuen Stapel
 2. Nimm jeweils nächste Karte des Originalstapels: füge diese an der richtigen Stelle in den neuen Stapel ein

Sortieren durch Einfügen: Beispiel



Sortieren durch Einfügen: Algorithmus

```
class InsertionSort implements Sort{  
  
    public void execute(int[] f) {  
        int m, j;  
        for (int i = 1; i < f.length; i++){  
            j = i;  
            m = f[i];  
            while (j > 0 && f[j-1] > m) {  
                /*verschiebe f[j-1] nach rechts  
                f[j] = f[j-1];  
                j--;  
            }  
            f[j] = m;  
        }  
    }  
}
```

f.length viele Elemente
im Array von Pos. 0
bis f.length-1

Wenn $f[j-1] > m$,
schiebe $f[j-1]$ nach
rechts

Setze $f[i]$ an
Position $f[j]$

Analyse

Theorem (Terminierung)

Der Algorithmus InsertionSort terminiert für jede Eingabe $\text{int}[]$ f nach endlicher Zeit.

Beweis

Die Laufvariable i in der äußeren for-Schleife wird in jedem Durchgang um eins erhöht und wird damit irgendwann die Abbruchbedingung (eine Konstante) erreichen. Die Laufvariable j der inneren while-Schleife wird in jedem Durchgang um eins verringert und somit die Schleifenbedingung $j > 0$ irgendwann nicht mehr erfüllen. \square

Analyse

Theorem (Korrektheit)

Der Algorithmus InsertionSort löst das Problem des vergleichsbasierten Sortierens.

Beweis:

Wir zeigen, dass die folgende Aussage eine *Invariante* der äußeren for-Schleife ist (d.h. sie ist am Ende eines jeden Schleifendurchgangs gültig):

Das Teilarray $f[0..i]$ ist sortiert

Damit gilt auch, dass nach Abbruch der for-Schleife das Array $f[0..n]=f$ (mit $n=f.length-1$) sortiert ist.

Analyse

Zu zeigen: Am Ende eines jeden Durchgangs der äußeren for-Schleife ist $f[0..i]$ sortiert.

Zeige dies durch Induktion nach i :

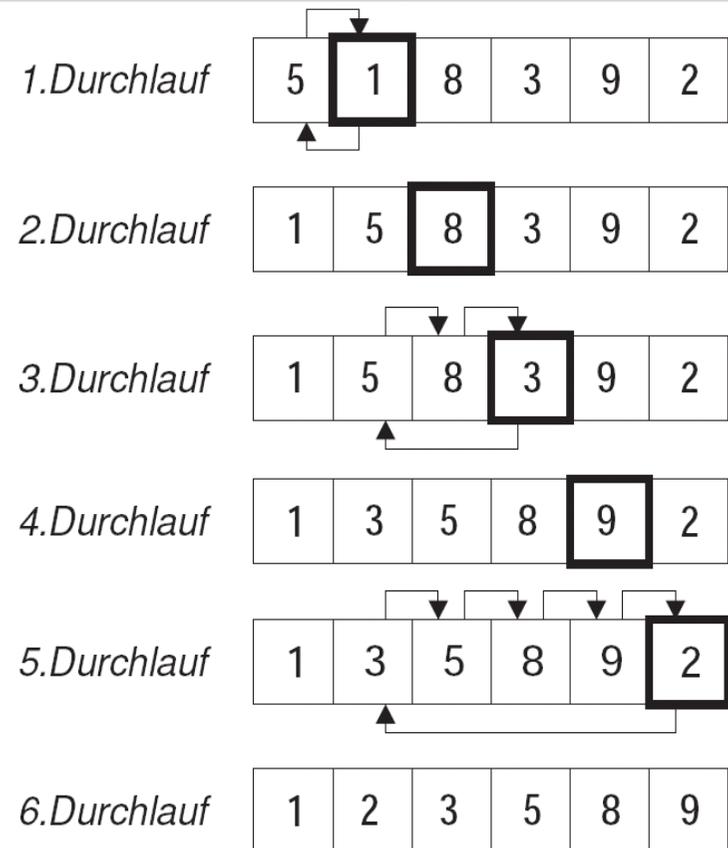
- $i=1$: Im ersten Durchgang wird das erste Element $f[0]$ mit dem zweiten Element $f[1]$ verglichen und ggfs. getauscht um Sortierung zu erreichen (while-Bedingung)

Analyse

- $i \rightarrow i+1$: Angenommen $f[0..i]$ ist am Anfang der äußeren for-Schleife im Durchgang $i+1$ sortiert. In der while-Schleife werden Elemente solange einen Platz weiter nach hinten verschoben, bis ein Index k erreicht wird, sodass alle Elemente mit Index $0..k-1$ kleiner/gleich dem ursprünglichen Element an Index $i+1$ sind (Induktionsbedingung) und alle Elemente mit Index $k+1..i+1$ größer sind (while-Bedingung). Das ursprüngliche Element an Index $i+1$ wird dann an Position k geschrieben. Damit gilt, dass $f[0..i+1]$ sortiert ist. \square

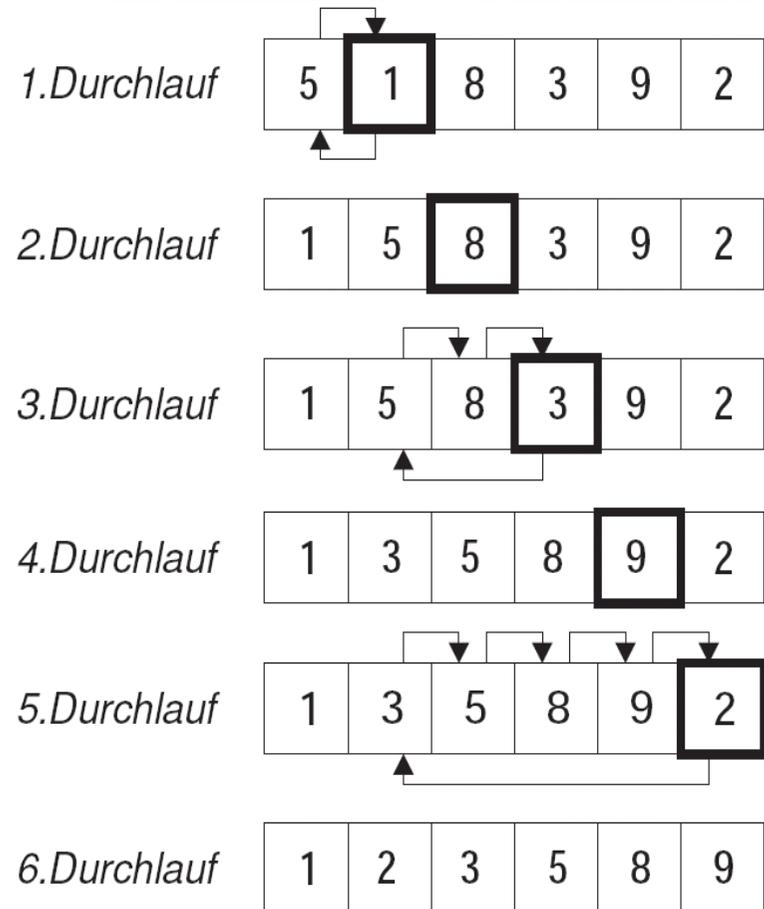
Aufwandsanalyse

- Anzahl der Vertauschungen
- Anzahl der Vergleiche
- Vergleiche dominieren Vertauschungen, d.h. es werden (wesentlich) mehr Vergleiche als Vertauschungen benötigt



Aufwandsanalyse (2)

- **Bester Fall**
 - ◆ Liste ist schon sortiert
- **Mittlerer (zu erwartender) Fall**
 - ◆ Liste unsortiert
- **Schlechtester Fall**
 - ◆ z.B.: Liste ist absteigend sortiert



Aufwandsanalyse (3)

- Wir müssen in jedem Fall alle Elemente $i:=1$ bis $n-1$ durchgehen, d.h. immer Faktor $n-1$ für die Anzahl der Vergleiche

$n = f.length$

- Dann müssen wir zur korrekten Einfügeposition zurückgehen

Bester Fall: Liste sortiert

- Einfügeposition ist gleich nach einem Schritt an Position $i-1$, d.h. Anzahl der Vergleiche = Anzahl der Schleifendurchläufe = $n-1$
- Bei jedem Rückweg zur Einfügeposition: Faktor 1

Gesamtzahl der Vergleiche: $(n-1) \cdot 1 = n-1$

Für große Listen abgeschätzt als $n-1 \approx n$

- „linearer Aufwand“

Aufwandsanalyse (4)

Mittlerer (zu erwartender) Fall: Liste unsortiert

- Einfügeposition wahrscheinlich auf Hälfte des Rückweges
- Aufwandsabschätzung
 - ◆ Bei jedem der $n-1$ Rückwege, addiere $(i-1)/2$ Vergleiche
 - ◆ Gesamtanzahl der Vergleiche:

$$\begin{aligned} & (n-1)/2 + (n-2)/2 + (n-3)/2 + \dots + 2/2 + 1/2 \\ = & \frac{(n-1) + (n-2) + (n-3) + \dots + 2 + 1}{2} \\ = & \frac{1}{2} \cdot \frac{n \cdot (n-1)}{2} \\ = & \frac{n \cdot (n-1)}{4} \approx \frac{n^2}{4} \end{aligned}$$

Gaußsche
Summenformel
für $k=1 \dots n-1$

Aufwandsanalyse (5)

Schlechtester Fall: z.B. Liste umgekehrt sortiert

- Einfügeposition am Endes des Rückwegs bei Position 1
- Bei jedem der $n-1$ Rückwege müssen $i-1$ Elemente verglichen werden (d.h. alle vorherigen Elemente $f[1...i-1]$)
- Analog zu vorhergehenden Überlegungen, bloß **doppelte** Rückweglänge

Daraus ergibt sich die Gesamtanzahl der Vergleiche:

$$(n - 1) + (n - 2) + (n - 3) + \dots + 2 + 1$$
$$= \frac{n \cdot (n - 1)}{2} \approx \frac{n^2}{2}$$

Gaußsche
Summenformel
für $k=1 \dots n-1$

Letzten beiden Fälle: „quadratischer Aufwand“, wenn konstante Faktoren nicht berücksichtigt werden

Aufwandsanalyse (6)

Theorem (Laufzeit)

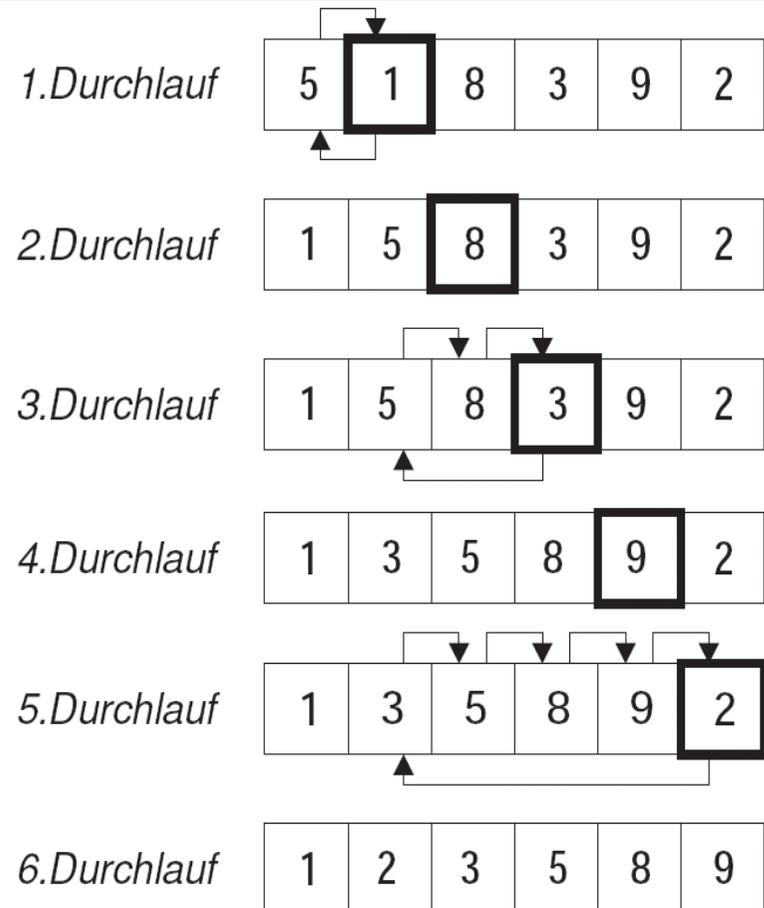
Die Anzahl der Vergleichsoperationen von InsertionSort ist im besten Fall $\Theta(n)$ und im durchschnittlichen und schlechtesten Fall $O(n^2)$.

Beweis

Siehe Vorüberlegungen. \square

Optimierung

- In der vorgestellten Version des Algorithmus wird die Einfügeposition eines Elements durch (umgekehrte) sequenzielle Suche gefunden
- Verwendet man hier binäre Suche (das Teilarray vor dem aktuellen Element ist sortiert!) kann die Anzahl der Vergleichsoperationen gesenkt werden zu $O(n \log n)$



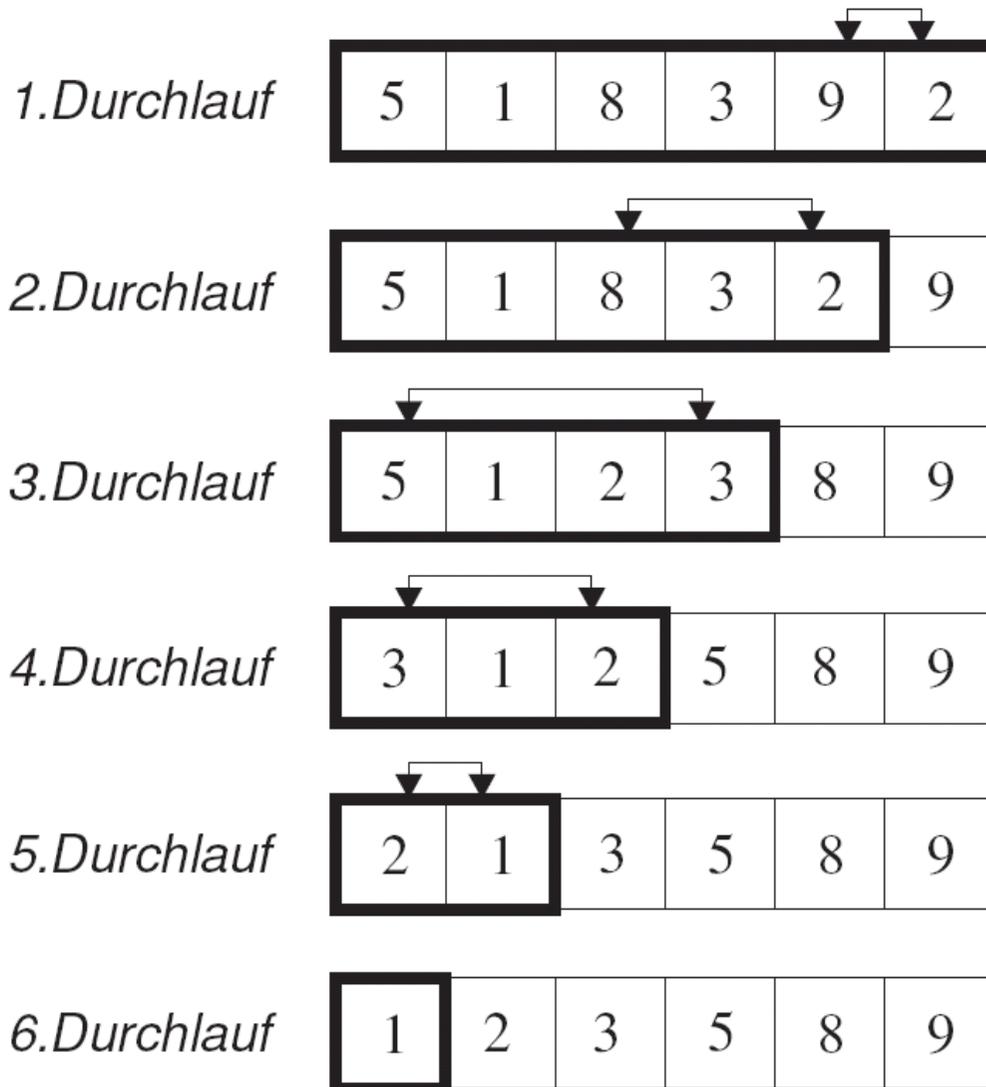
Algorithmen und Datenstrukturen

6.1.2 SELECTION SORT

Sortieren durch Selektion

- Idee
 - ◆ Suche jeweils größten Wert im Array und tausche diesen an die letzte Stelle
 - ◆ Fahre dann mit der um 1 kleineren Liste fort

Sortieren durch Selektion: Beispiel



Sortieren durch Selektion

- Idee: Suche jeweils größten Wert und tausche diesen an die letzte Stelle, fahre dann mit der um 1 kleineren Liste fort

```
class SelectionSort implements Sort{
    void execute (int[] f) {
        int marker = f.length -1;
        while (marker >= 0) {
            /*bestimme größtes Element links v. Marker*/
            int max = 0; /* Indexposition*/
            for (int i = 1; i <= marker; i++){
                if (f[i] > f[max])
                    max = i;
            }
            swap(f, marker, max);
            marker--; /*verkleinere Array */
        }
    }
}
```

Sortieren durch Selektion (2)

- Hilfsmethode swap
 - ◆ vertausche zwei Elemente im Array

```
void swap(int[] f, int idx1, int idx2) {  
    int tmp = f[idx1];  
    f[idx1] = f[idx2];  
    f[idx2] = tmp;  
}
```

Analyse

Theorem (Terminierung)

Der Algorithmus SelectionSort terminiert für jede Eingabe `int[] f` nach endlicher Zeit.

Beweis

Die Variable `marker` wird zu Anfang des Algorithmus auf einen positiven endlichen Wert gesetzt und in jedem Durchgang der äußeren `while`-Schleife um 1 verkleinert. Abbruch der `while`-Schleife erfolgt, wenn `marker` kleiner 0 ist, also wird die `while`-Schleife endlich oft durchlaufen. Die innere `for`-Schleife hat in jedem Durchgang `marker`-viele (also endlich viele) Durchläufe.



Analyse

Theorem (Korrektheit)

Der Algorithmus SelectionSort löst das Problem des vergleichsbasierten Sortierens.

Beweis:

Übung

Analyse

Theorem (Laufzeit)

Der Algorithmus SelectionSort hat eine Laufzeit von $\Theta(n^2)$. Diese Laufzeit ist dieselbe für den besten, mittleren und schlechtesten Fall.

Beweis:

Die äußere while-Schleife wird genau n -mal ($n=f.length$) durchlaufen. Dort werden somit n Vertauschungen vorgenommen (=jeweils konstanter Aufwand). Die innere for-Schleife hat im i -ten Durchlauf der while-Schleife $n-i$ Durchläufe mit jeweils einem Vergleich, deswegen insgesamt

$$\begin{aligned} & (n - 1) + (n - 2) + (n - 3) + \dots + 1 \\ &= \frac{n \cdot (n - 1)}{2} \approx \frac{n^2}{2} \end{aligned}$$

viele Vergleiche, also $\Theta(n^2)$. \square

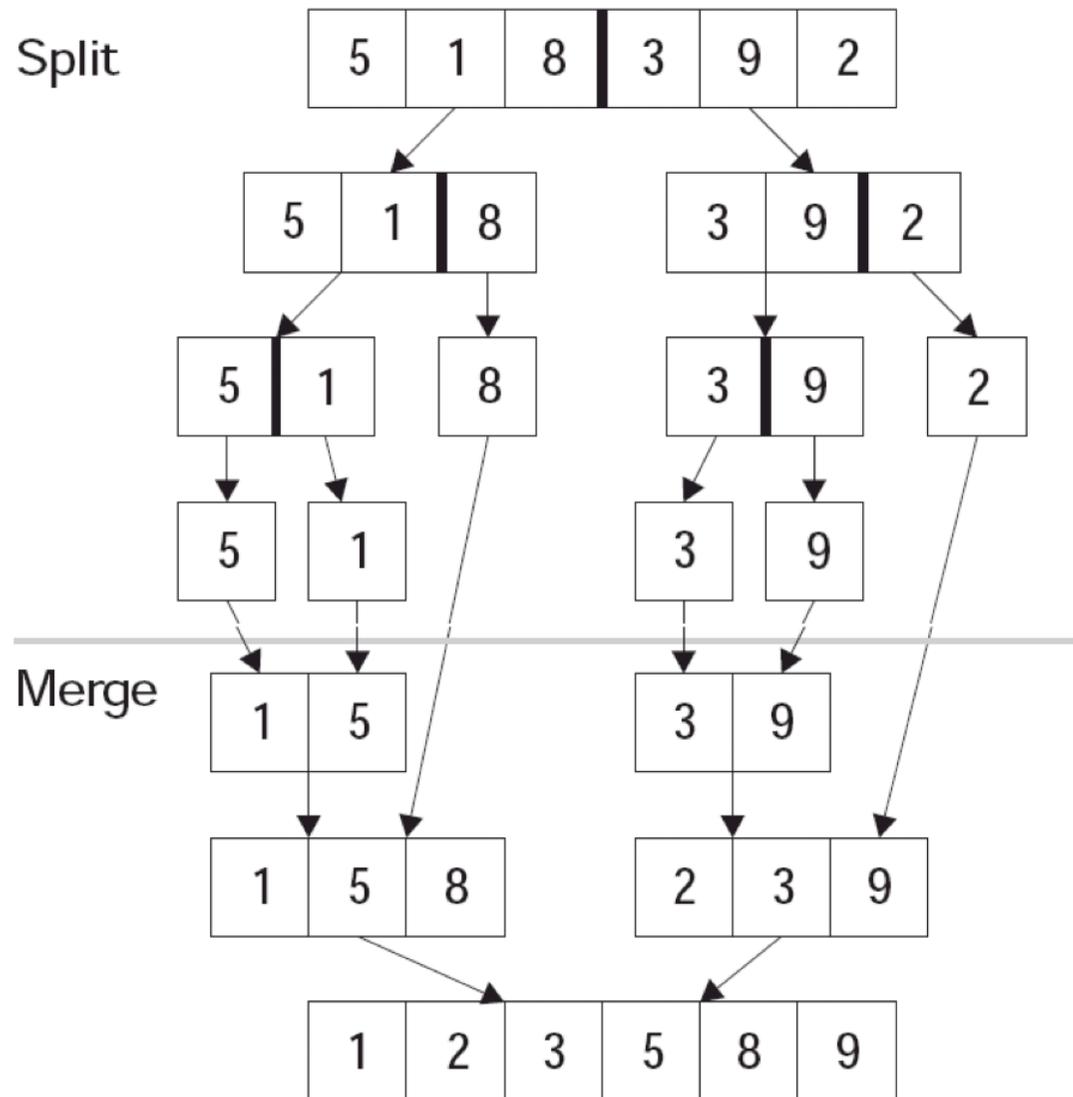
Algorithmen und Datenstrukturen

6.1.3 MERGE SORT

MergeSort: Prinzip

- MergeSort ist ein Divide-and-Conquer Algorithmus zum vergleichsbasierten Sortieren
- Idee:
 - 1. Teile** die zu sortierende Folge in zwei Teile
 - 2. Sortiere** beide Teile getrennt voneinander
 - 3. Mische** die Teilergebnisse in der richtigen Reihenfolge zusammen

Beispiel



Algorithmus

```
class MergeSort implements Sort{

    void execute(int[] f) {
        int[] tmpF = new int[f.length];
        mergeSort(f, tmpF, 0, f.length -1);
    }

    private void mergeSort(int[] f, int[] tmpF, int left,int right){
        if (left < right) {
            int m = (left + right)/2;
            mergeSort(f, tmpF, left, m);
            mergeSort(f, tmpF, m+1, right);
            merge(f, tmpF, left, m+1, right);
        }
    }

    ...
}
```

Algorithmus (2) - merge

```
...  
  
private void merge(int[] f, int[] tmpF, int startLeft,  
    int startRight, int endRight) {  
  
    int endLeft = startRight-1;  
    int tmpPos = startLeft;  
    int numElements = endRight - startLeft +1;  
    while (startLeft <= endLeft && startRight <= endRight)  
        if (f[startLeft] < f[startRight])  
            tmpF[tmpPos++] = f[startLeft++];  
        else  
            tmpF[tmpPos++] = f[startRight++];  
  
    while (startLeft <= endLeft)  
        tmpF[tmpPos++] = f[startLeft++];  
    while (startRight <= endRight)  
        tmpF[tmpPos++] = f[startRight++];  
  
    for (int i = 0; i < numElements; i++, endRight--)  
        f[endRight] = tmpF[endRight];  
}  
  
}
```

Analyse

Theorem (Terminierung)

Der Algorithmus MergeSort terminiert für jede Eingabe `int[] f` nach endlicher Zeit.

Beweis

Zeige zunächst, dass jeder Aufruf `mergeSort(int[] f, int[] tmpF, int left, int right)` terminiert:

- Falls `left < right` nicht gilt, terminiert der Aufruf sofort
- Andernfalls rufen wir `mergeSort` rekursiv auf, wobei entweder `left` einen echt größeren oder `right` einen echt kleineren Wert erhält. In jedem Fall wird nach einem gewissen rekursiven Abstieg irgendwann `left < right` nicht mehr gelten.

Terminierung der Methode `merge` ist Übung.

Analyse

Theorem (Korrektheit)

Der Algorithmus MergeSort löst das Problem des vergleichsbasierten Sortierens.

Beweis:

Durch Induktion nach $n = f.length$. Annahme $n=2^k$ für eine ganze Zahl k .

- $n=1$: Für $n=1$ ist der erste Aufruf der mergeSort Hilfsmethode `mergeSort(f, tmpF, 0, 0)` und somit gilt nicht $left < right$. Die Methode terminiert ohne Änderung an f . Dies ist korrekt, da jedes einelementige Array sortiert ist.

Analyse

- $n/2 \rightarrow n$: Sei $f[0\dots n-1]$ ein beliebiges Array. Der erste Aufruf $\text{mergeSort}(f, \text{tmpF}, 0, n-1)$ erfüllt $\text{left} < \text{right}$ und es werden folgende Rekursive Aufrufe getätigt:

$\text{mergeSort}(f, \text{tmpF}, 0, n/2-1)$

$\text{mergeSort}(f, \text{tmpF}, n/2, n-1)$

Beide Aufrufe erhalten ein Array der Länge $n/2$. Nach Induktionsannahme gilt, dass anschliessend sowohl $f[0\dots n/2-1]$ als auch $f[n/2\dots n-1]$ separat sortiert sind.

Noch zu zeigen ist, dass merge korrekt zwei sortierte Arrays in ein sortiertes Array mischt (Übung) \square

Analyse

Theorem (Laufzeit)

Der Algorithmus MergeSort hat eine Laufzeit von $\Theta(n \log_2 n)$. Diese Laufzeit ist dieselbe für den besten, mittleren und schlechtesten Fall.

Beweis:

Erste Beobachtung: die Methode `merge` hat eine Laufzeit von $\Theta(n)$ wobei n die Länge des gemischten Arrays ist (Übung). Nun kann man leicht eine Rekursionsgleichung für die Laufzeit von MergeSort $T(n)$ angeben:

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{sonst} \end{cases}$$

Analyse

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{sonst} \end{cases}$$

Anwendung des Master-Theorems:

2. Fall: Wenn $f(n) \in \Theta(n^{\log_b a} \cdot ld^k n)$ für ein $k \geq 0$
dann $T(n) = \Theta(n^{\log_b a} \cdot ld^{k+1} n)$

Hier ist $a=2$ und $b=2$ und es folgt $n^{\log_b a} = n^{\log_2 2} = n^1 = n$.

Es ist zudem $f(n)=n$ und es gilt für $k=0$:

$$n \in \Theta(n ld^k n) = \Theta(n)$$

Es folgt $T(n) \in \Theta(n ld^{k+1} n) = \Theta(n ld n)$. \square

Algorithmen und Datenstrukturen

6.1.4 QUICK SORT

QuickSort: Prinzip

Idee:

- Divide-and-Conquer und rekursive Aufteilung (wie bei MergeSort)
- Aber: Vermeidung des Mischvorgangs (speicherintensiv!)
 1. Wähle ein Element der Liste aus (z.B. das letzte), dieses nennen wir *Pivot-Element*
 2. Bestimme die Position, die das Pivot-Element in der vollständig sortierten Liste haben muss
 3. Bringe das Pivot-Element an diese Position und stelle sicher, dass alle kleineren Elemente links und alle größeren Element rechts von ihm stehen
 4. Fahre rekursiv mit den Teillisten rechts und links fort

QuickSort: Beispiel

f:

8	9	2	6	7	3	4	1	5
---	---	---	---	---	---	---	---	---

Wähle $f[8]=5$ als Pivot-Element

f:

8	9	2	6	7	3	4	1	5
---	---	---	---	---	---	---	---	---

Suche von links an Elemente, die kleiner dem Pivot-Element sind

f:

8	9	2	6	7	3	4	1	5
---	---	---	---	---	---	---	---	---

↑

Vertausche mit dem ersten größeren Element

f:

2	9	8	6	7	3	4	1	5
---	---	---	---	---	---	---	---	---

↑

Suche das nächste kleinere Element

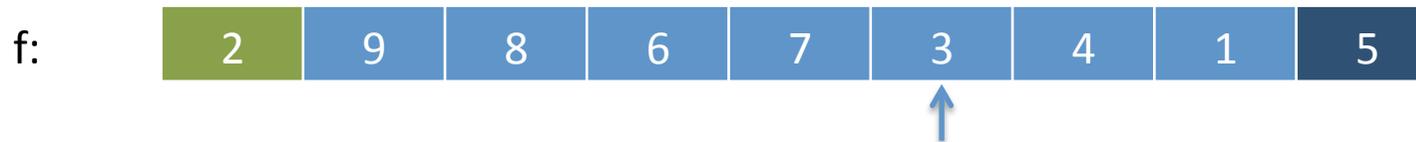
f:

2	9	8	6	7	3	4	1	5
---	---	---	---	---	---	---	---	---

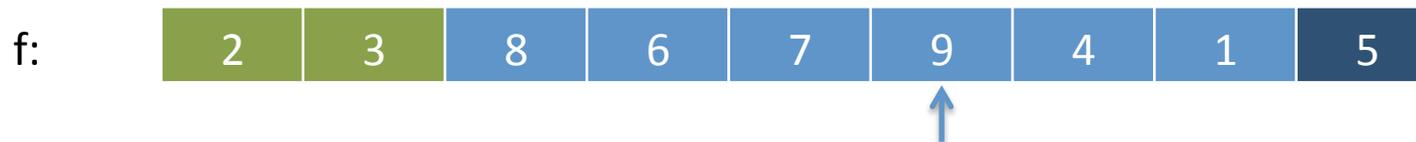
↑

QuickSort: Beispiel

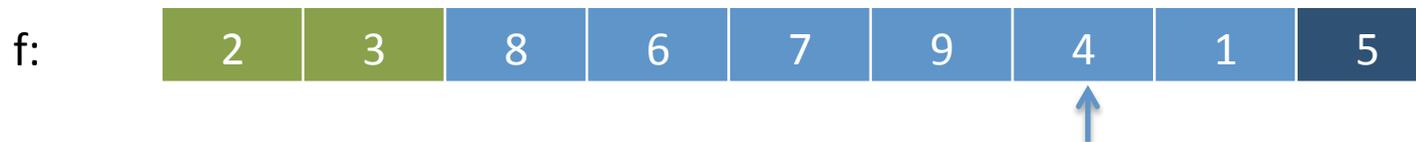
Suche das nächste kleinere Element



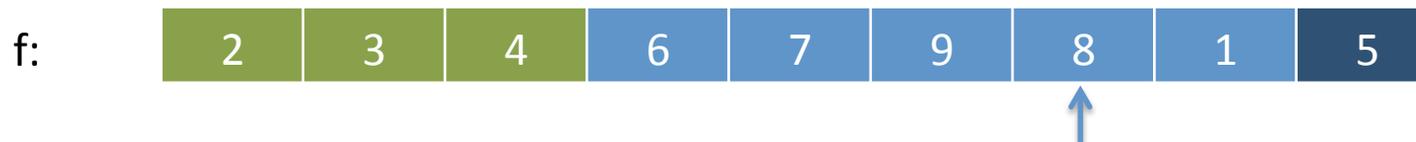
Vertausche mit dem zweiten größeren Element



Suche das nächste kleinere Element

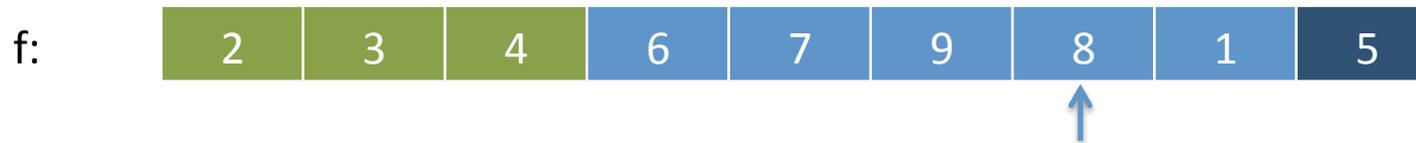


Vertausche mit dem dritten größeren Element

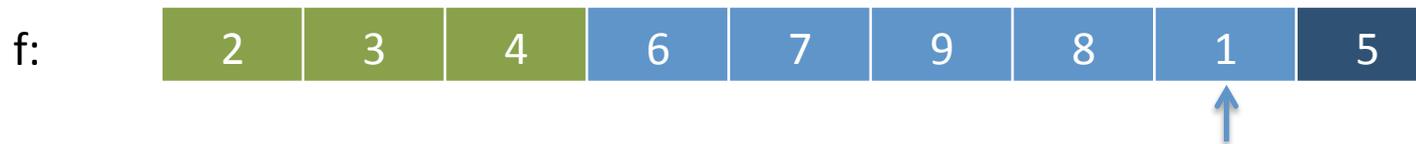


QuickSort: Beispiel

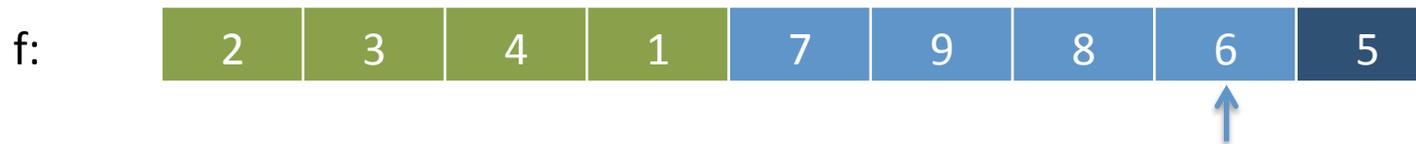
Vertausche mit dem dritten größeren Element



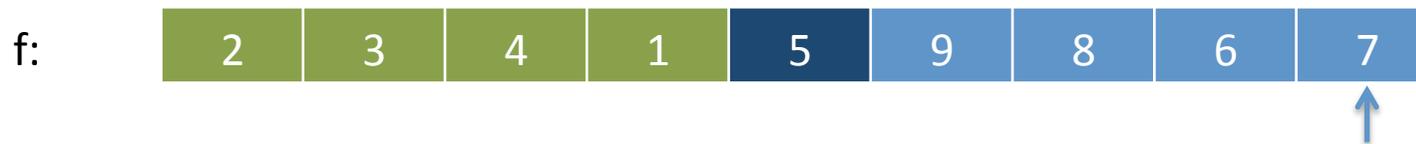
Suche das nächste kleinere Element



Vertausche mit dem vierten größeren Element



Ist man rechts angekommen, bringe das Pivot-Element an die richtige Position, d.h. tausche mit dem fünften größeren Element



QuickSort: Beispiel

f:

2	3	4	1	5	9	8	6	7
---	---	---	---	---	---	---	---	---

- Beobachtungen:
 - Das Pivot-Element steht an seiner finalen Position
 - Alle Elemente links vom Pivot-Element sind kleiner
 - Alle Elemente rechts vom Pivot Element sind größer
- Nun rekursiver Abstieg:
 - Sortiere

2	3	4	1
---	---	---	---
 - Sortiere

9	8	6	7
---	---	---	---

Sortierprinzip

- Sortieren einer Folge $f[u\dots o]$ nach dem „divide-and-conquer“-Prinzip
 - **Divide:** Teile Folge $f[u\dots o]$ in zwei Teilfolgen $f[u\dots p-1]$ und $f[p+1\dots o]$ mit der Eigenschaft:
 - ◆ $f[i] \leq f[p]$ für alle $i = u, \dots, p-1$
 - ◆ $f[i] > f[p]$ für alle $i = p+1, \dots, o$
 - **Conquer:** Sortiere Teilfolgen $f[u\dots p-1]$ und $f[p+1\dots o]$
 - **Combine:** Verbinde Teilfolgen zu $f[u\dots o]$
 - ◆ Keine Vergleiche erforderlich, da Teilfolgen bereits sortiert sind

Das Pivot-Element

- Im Prinzip muss man nicht das letzte Element als Pivot-Element wählen
- Je nach Verteilung der Daten, kann es sinnvoll sein ein anderes Element zu wählen
 - Wenn beispielsweise die Liste schon fast sortiert ist, sollte man immer das mittlere Element wählen
 - Eine optimale Rekursion erhält man, wenn man immer den Median als Pivot-Element wählt (dieser ist aber nicht direkt bestimmbar, dafür müsste man die Liste erst sortiert haben)
- Hat man ein Pivot-Element ausgewählt, tauscht man dies einfach mit dem letzten Element und benutzt den Algorithmus wie zuvor

Algorithmus

```
class QuickSort implements Sort{  
  
    void execute(int[] f){  
        quickSort(f, 0, f.length-1);  
    }  
  
    void quickSort(int[] f, int u, int o) {  
        if (u < o) {  
            int p = o;  
            int pn = split(f,u,o,p);  
            quickSort(f,u,pn-1);  
            quickSort(f,pn+1,o);  
        }  
    }  
  
    ...  
}
```

p:
Gibt an, an welcher
Position das Pivot-
Element ist.
Hier: o
Genauso u, das mittlere,
oder jedes andere
Element möglich!

Algorithmus

```
...
int split(int[] f, int u, int o, int p) {
    int pivot = f[p];
    swap(f, p, o);
    int pn = u;
    for (int j = u; j < o; j++) {
        if (f[j] <= pivot) {
            swap(f, pn, j);
            pn++;
        }
    }
    swap(f, pn, o);
    return pn;
}

void swap(int[] f, int x, int y){
    int tmp = f[x];
    f[x] = f[y];
    f[y] = tmp;
}
}
```

Tausche Pivot-Element
mit dem letzten Element
(kann entfallen wenn
immer $p=o$ gewählt wird)

Bringe kleinere Elemente nach
vorne und größere nach hinten

Bringe das Pivot-Element an
die richtige Position und gebe
diese zurück

Hilfsmethode zum Vertauschen

Analyse

Theorem (Terminierung)

Der Algorithmus quickSort terminiert für jede Eingabe `int[] f` nach endlicher Zeit.

Beweis

In jedem rekursiven Aufruf von quickSort ist die Eingabelänge um mindestens 1 kleiner als vorher und die Rekursionsanfang ist erreicht wenn die Länge gleich 1 ist. In der Methode `split` gibt es nur eine for-Schleife, dessen Zähler `j` in jedem Durchgang inkrementiert wird. Da `u < o` wird die for-Schleife also nur endlich oft durchlaufen. \square

Analyse

Theorem (Korrektheit)

Der Algorithmus quickSort löst das Problem des vergleichsbasierten Sortierens.

Beweis:

Die Korrektheit der Methode `swap` ist zunächst offensichtlich.

Zeige nun, dass nach Aufruf $p_n = \text{split}(f, u, o, p)$ für $u < o$ und $p \in [u, o]$ gilt:

1. $f[p]$ wurde zu $f[p_n]$ verschoben
2. $f[i] \leq f[p_n]$ für $i = u, \dots, p_n - 1$
3. $f[i] > f[p_n]$ für $i = p_n + 1, \dots, o$

Analyse

1. $f[p]$ wurde zu $f[pn]$ verschoben

Dies ist klar (vorletzte Zeile der Methode split)

2. $f[i] \leq f[pn]$ für $i=u, \dots, pn-1$

pn wird zu anfangs mit u initialisiert und immer dann inkrementiert, wenn die Position $f[pn]$ durch ein Element, das kleiner/gleich dem Pivot-Element ist, belegt wird.

3. $f[i] > f[pn]$ für $i=pn+1, \dots, o$

Folgt aus der Beobachtung, dass in 2.) immer „genau dann“ gilt.

Beachte zudem, dass Element immer getauscht werden, also die Elemente im Array stets dieselben bleiben.

Analyse

Die Korrektheit der Methode quickSort folgt nach Induktion nach der Länge von f ($n=f.length$):

- $n=1$: Der Algorithmus terminiert sofort und ein einelementiges Array ist stets sortiert
- $n \rightarrow n+1$: Nach Korrektheit von split steht das Pivot-Element an der richtigen Stelle und links und rechts stehen jeweils nur kleinere/größere Element. Die beiden rekursiven Aufrufe von quickSort erhalten jeweils ein Array, das echt kleiner als $n+1$ ist (mindestens das Pivot-Element ist nicht mehr Teil des übergebenen Arrays). Nach Induktionsannahme folgt die Korrektheit von quickSort. \square

Analyse

Theorem (Laufzeit)

Wird als Pivot-Element stets der Median des aktuell betrachteten Arrays gewählt, so hat der Algorithmus quickSort eine Laufzeit von $\Theta(n \log n)$.

Beweis:

Es gilt zunächst, dass $\text{split} \in \Theta(n)$ (mit $n = o - u$): ausschlaggebend ist hier die for-Schleife, die genau n -mal durchlaufen wird.

Analyse

Gilt nach dem Aufruf von `split` stets $p_n = (u+o)/2$ (dies ist gleichbedeutend damit, dass das Pivot-Element stets der Median ist), so erhalten wir folgende Rekursionsgleichung für `quickSort`:

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2T((n-1)/2) + \Theta(n) & \text{sonst} \end{cases}$$

Dies ist (fast) dieselbe Rekursionsgleichung wie für `MergeSort` und es folgt $T(n) \in \Theta(n \log n)$. \square

Analyse

- Was ist wenn die Voraussetzung des Theorems nicht erfüllt ist?

- Beispiel:



- ungleiche Rekursionsaufrufe

Analyse

Theorem (Laufzeit)

Im schlechtesten Fall hat der Algorithmus quickSort eine Laufzeit von $\Theta(n^2)$.

Beweis:

Angenommen, die Aufteilung erzeugt ein Teilarray mit Länge $n-1$ und ein Teilarray mit Länge 0 (Pivot-Element ist also immer Minimum oder Maximum), dann erhalten wir folgende Rekursionsgleichung für die Laufzeit:

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + \Theta(n) & \text{sonst} \end{cases}$$

Durch Induktionsbeweis kann leicht gezeigt werden, dass $T(n) \in \Theta(n^2)$. Dies ist auch tatsächlich der schlechteste Fall (Übung). \square

Analyse

- Für den mittleren Fall kann gezeigt werden, dass quickSort einen Aufwand von $\Theta(n \log n)$ hat (wie im besten Fall), die in Θ versteckten Konstanten sind nur etwas größer

Algorithmen und Datenstrukturen

6.1.5 ALLGEMEINE BETRACHTUNGEN

Eigenschaften der betrachteten Algorithmen

- Komplexität im Durchschnittsfall und im Schlechtesten Fall ist nie besser als $n \cdot \log n$
- Sortierung erfolgt ausschließlich durch Vergleich der Eingabe-Elemente (*comparison sorts*) – *vergleichsorientierte Sortierverfahren*
- Wir zeigen nun: $n \cdot \log n$ Vergleiche ist eine untere Schranke für „Comparison Sort“-Algorithmen
 - ◆ Dies heißt dann, dass Sortieralgorithmen mit Komplexität (schlechtester Fall) von $n \cdot \log n$ (z.B. MergeSort) asymptotisch optimal sind.

Untere Schranke

- Problembeschreibung

- ◆ Eingabe: $\langle a_1, a_2, \dots, a_n \rangle$

- ◆ Vergleichstests: $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j, a_i > a_j$

- Vereinfachende Annahme:

- ◆ Verschiedene Elemente: $a_i = a_j$ entfällt

- ◆ Restliche Tests: $a_i < a_j, a_i \leq a_j, a_i \geq a_j, a_i > a_j$

- Liefern alle gleichwertige Information

- Sie bestimmen Reihenfolge von a_i und a_j

- Können uns beschränken auf: $a_i \leq a_j$

- Binäre Entscheidung, es gilt:

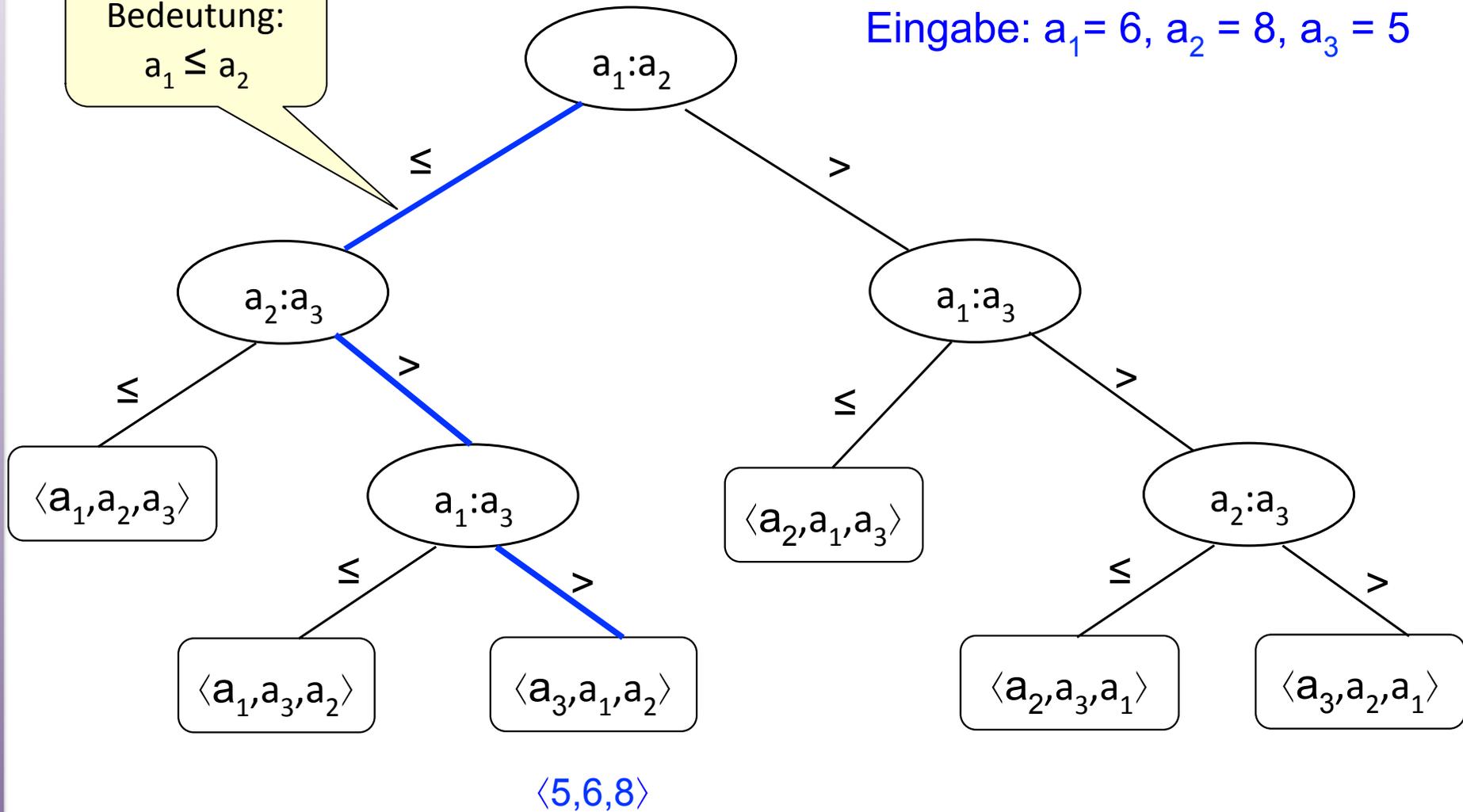
- (1) entweder $a_i \leq a_j$

- (2) oder $a_i > a_j$

Entscheidungsbaum

Bedeutung:
 $a_1 \leq a_2$

Beispiel:
 Eingabe: $a_1 = 6, a_2 = 8, a_3 = 5$



Entscheidungsbaum - Beobachtungen

- Innere Knoten vergleichen Elemente a_i und a_j
 - ◆ Test ob $a_i \leq a_j$ gilt oder nicht
- Blätter: Permutationen $\langle \pi(a_1), \dots, \pi(a_n) \rangle$
- Sortieren: Finden einen Pfad von der Wurzel zu einem Blatt
 - ◆ An jedem internen Knoten erfolgt ein Vergleich
 - Entsprechend links oder rechts weiter suchen
 - ◆ Ist ein Blatt erreicht: Sortieralgorithmus hat eine Ordnung / Permutation der Elemente erstellt

Folgerung

- Jeder Sortieralgorithmus muss jede Permutation der n Eingabe-Elemente erreichen $\rightarrow n!$
 - $n!$ Blätter (die alle von der Wurzel erreichbar sind)
 - Andernfalls kann er zwei unterschiedliche Eingaben nicht unterscheiden und liefert für beide dasselbe Ergebnis (eins davon muss somit falsch klassifiziert sein)
- Anzahl an Vergleichen (schlechtesten Fall) ist die Pfadlänge von Wurzel bis Blatt (Höhe des Baums)

Theorem:

Jeder vergleichsorientierte Sortieralgorithmus benötigt im schlechtesten Fall mindestens $n \cdot \log n$ Vergleiche.

Folgerung (2)

Beweis

- Gegeben:
 - ◆ Sei n die Anzahl der Elemente
 - ◆ Sei Pfadlänge / Höhe des Baums: h
 - ◆ Sei die Anzahl von Blättern: b
- Jede Permutation muss in einem Blatt sein: $n! \leq b$
- Binärbaum der Höhe h hat max. 2^h Blätter
 - $n! \leq b \leq 2^h$
- Logarithmieren:
$$h \geq \log_2(n!) \approx n \cdot \log_2(n)$$

(genauer = $\Omega(n \cdot \log_2(n))$)

Hinweis: folgt aus Stirlings Approximation von $n!$

Algorithmen und Datenstrukturen

6.1 ALGORITHMEN FÜR VERGLEICHSBASIERTES SORTIEREN

ZUSAMMENFASSUNG

Zusammenfassung

- InsertionSort
 - Bester Fall: $\Theta(n)$
 - Durchschnittlicher und schlechtester Fall: $O(n^2)$
- SelectionSort
 - Bester, durchschnittlicher und schlechtester Fall: $\Theta(n^2)$
- MergeSort
 - Bester, durchschnittlicher und schlechtester Fall: $\Theta(n \log_2 n)$
- QuickSort
 - Bester und durchschnittlicher Fall: $\Theta(n \log n)$
 - Schlechtester Fall: $O(n^2)$
- Allgemeine untere Schranke für vergleichsbasierte Sortierverfahren (schlechtester Fall): $n \cdot \log n$

Algorithmen und Datenstrukturen

6.2 WEITERE SORTIERPROBLEME

Rückblick und Ausblick

- Bisherige Verfahren:
 - ◆ Sortieren n Zahlen mit $n \cdot \log n$ Vergleichen
 - z.B. QuickSort im Durchschnittsfall
 - ◆ Bestimmen Sortierung durch Vergleiche / Vergleich-Sortierung
 - ◆ Wir haben gezeigt: $n \log n$ ist eine untere Schranke für Sortieralgorithmen die auf Vergleiche/Elementvergleiche basieren
- Geht es besser?
 - Wenn wir weitere Annahmen machen, ja

Algorithmen und Datenstrukturen

6.2.1 NICHT-VERGLEICHSBASIERTES SORTIEREN

Bucket-Sort

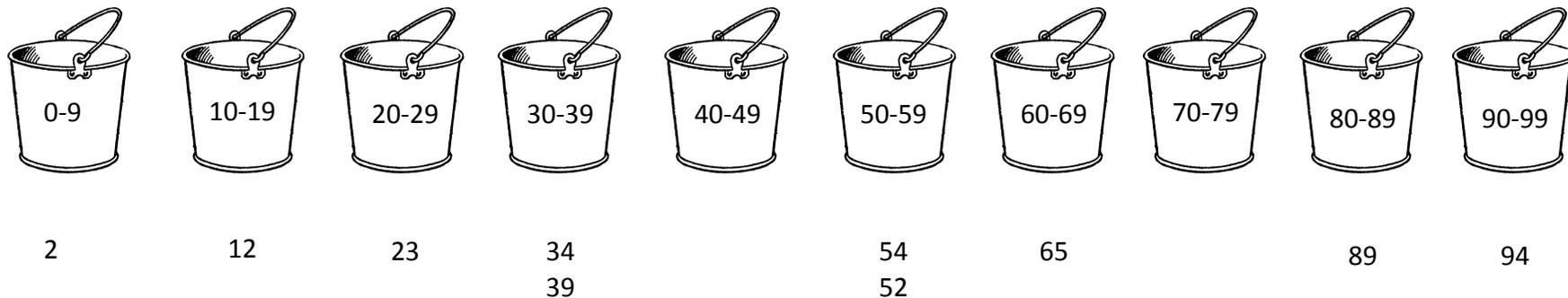
- Annahme:
 - Elemente kommen aus einem vorher festgelegtem Bereich, z.B. $[0..K)$ für eine natürliche Zahl K .
 - Sie sind in $[0..K)$ ungefähr gleich verteilt

Grundidee BucketSort (bei einer Eingabe von n Zahlen):

1. Erzeuge n verkettete Listen (buckets), die das Intervall $[0..K)$ in n Teilintervalle der Größe K/n einteilen
 - Dies bedeutet: Zerlege Wertebereich in n gleich große Teilintervalle (buckets)
2. Füge jedes Element in die passende Liste ein (in ein bucket)
3. Verkette die Listen

Bucket-Sort: Beispiel

- Annahme $K=100$
- Eingabe [23,54,2,12,65,34,39,94,89,52]



- Von links nach rechts, verkette Listen (wende evtl. vergleichbasiertes Sortierverfahren auf Buckets mit mehr als einem Element an)
- Ausgabe: [2,12,23,34,39,52,54,65,89,94]

Algorithmus

```
class BucketSort implements Sort{

    private int bound;
    public BucketSort(int bound){ this.bound = bound; }

    void execute(int[] f){
        int n = f.length;
        List<List<Integer>> buckets = createBuckets(n);
        for(int i = 0; i < n; i++){
            buckets.get(n*f[i]/bound).add(f[i]);
        }
        int idx = 0;
        for(int i = 0; i < n; i++){
            quickSort(buckets.get(i));
            for(int j = 0; j < buckets.get(i).size(); j++){
                f[idx++] = buckets.get(i).get(j);
            }
        }
    }

    List<List<Integer>> createBuckets(int n){
        List<List<Integer>> buckets
            = new ArrayList<List<Integer>>();
        for(int i = 0; i < n; i++){
            buckets.add(new ArrayList<Integer>());
        }
        return buckets;
    }
}
```

Analyse

Theorem (Terminierung)

Der Algorithmus BucketSort terminiert für jede Eingabe $\text{int}[]$ f nach endlicher Zeit.

Beweis

Alle Schleifenvariablen haben wohl-definierte Start- und Endpunkte. \square

Analyse

Theorem (Korrektheit)

Der Algorithmus BucketSort sortiert das Array f .

Beweis

Sei $i < j$ beliebig und betrachte $f[i]$ und $f[j]$ nach Anwendung $\text{execute}(f)$. Falls $n \cdot f[i] / \text{bound} = n \cdot f[j] / \text{bound}$ so sind $f[i]$ und $f[j]$ im selben Bucket gelandet. Nach Annahme ist QuickSort korrekt und hat den Bucket korrekt sortiert. Anschliessend wurde zunächst $f[i]$ und dann $f[j]$ in das Array f geschrieben, also folgt $f[i] \leq f[j]$. Fall $n \cdot f[i] / \text{bound} \neq$

$n \cdot f[j] / \text{bound}$ so muss $n \cdot f[i] / \text{bound} < n \cdot f[j] / \text{bound}$ gelten (da $i < j$), daraus folgt direkt $f[i] < f[j]$. Also ist f sortiert. \square

Analyse

Theorem (Laufzeit)

Ist f mit $n=f\text{-length}$ ein Array mit Elementen, die gleichverteilt sind in $[1..\text{bound}]$ so hat BucketSort auf f eine Laufzeit von $\Theta(n)$.

Beweis (Skizze)

Im Schnitt landet in jedes Bucket genau ein Element (da Gleichverteilung) und die Anwendung des Sortieralgorithmus auf jedes Bucket (und die innerste for-Schleife) kann vernachlässigt werden. Alle anderen Schleifen haben genau n Durchläufe, also $\Theta(n)$.



Analyse

- Falls die Annahme der Gleichverteilung nicht gilt, wird keine lineare Komplexität erreicht.
 - ◆ Wie wäre die Komplexität im schlechtesten Fall (bei Verletzung der Annahme)?

Algorithmen und Datenstrukturen

6.2.2 VERTEILTES SORTIEREN

Verteiltes Sortieren

- Eine Laufzeit von $n \log n$ für Sortieren ist sehr effizient für viele Anwendungen
- Das häufige Sortieren großer Mengen von Elementen kann dennoch ein Problem sein
 - Häufige Festplattenzugriffe
- Ausnutzung paralleler Hardware kann den Sortierprozess beschleunigen
- Algorithmen wie MergeSort bieten sich zur Verteilung an
 - Problem: Merge-Schritt ist nicht direkt parallelisierbar

Wir schauen uns verteiltes Sortieren exemplarisch am Batchersort-Algorithmus an

BatcherSort: Idee 1/2

- Grundsätzliche Idee wie bei MergeSort:
 - Teile das Array in zwei gleich große Teilarrays
 - Fahre rekursiv mit jedem Teilarray fort und sortiere
 - Mische die resultierenden Teilarrays
- Der Merge-Schritt wird jedoch bei BatcherSort auch rekursiv durchgeführt

BatcherSort: Idee 2/2

- Grundidee von BatcherSort:

- Sei l eine Liste mit Elementen $1, \dots, n$

9	2	5	1	8	7	3	6
---	---	---	---	---	---	---	---

- Sortiere Elemente $1, \dots, n/2$ und $n/2+1, \dots, n$

1	2	5	9	3	6	7	8
---	---	---	---	---	---	---	---

- Sortiere Elemente $1, 3, 5, \dots, n-1$ und $2, 4, 6, \dots, n$

1	2	3	6	5	8	7	9
---	---	---	---	---	---	---	---

- Vergleiche/vertausche Elemente $2/3, 4/5, 6/7$

1	2	3	5	6	7	8	9
---	---	---	---	---	---	---	---

- Liste ist anschliessend sortiert
- Annahme: Listen haben Länge $n=2^N$ für natürliche Zahl N

Algorithmus 1/2

```
class Batchersort implements Sort{  
  
    void execute(int[] list){  
        batchersort(list, 0, list.length-1);  
    }  
  
    void batchersort(int[] list, int u, int o){  
        if(u==o) return;  
        batchersort(list, u, (u+o)/2);  
        batchersort(list, (u+o)/2+1,o);  
        merge(list,u,o,1);  
    }  
    ...  
}
```

Kann parallel
ausgeführt werden

Sortierphase (ähnlich) wie in MergeSort

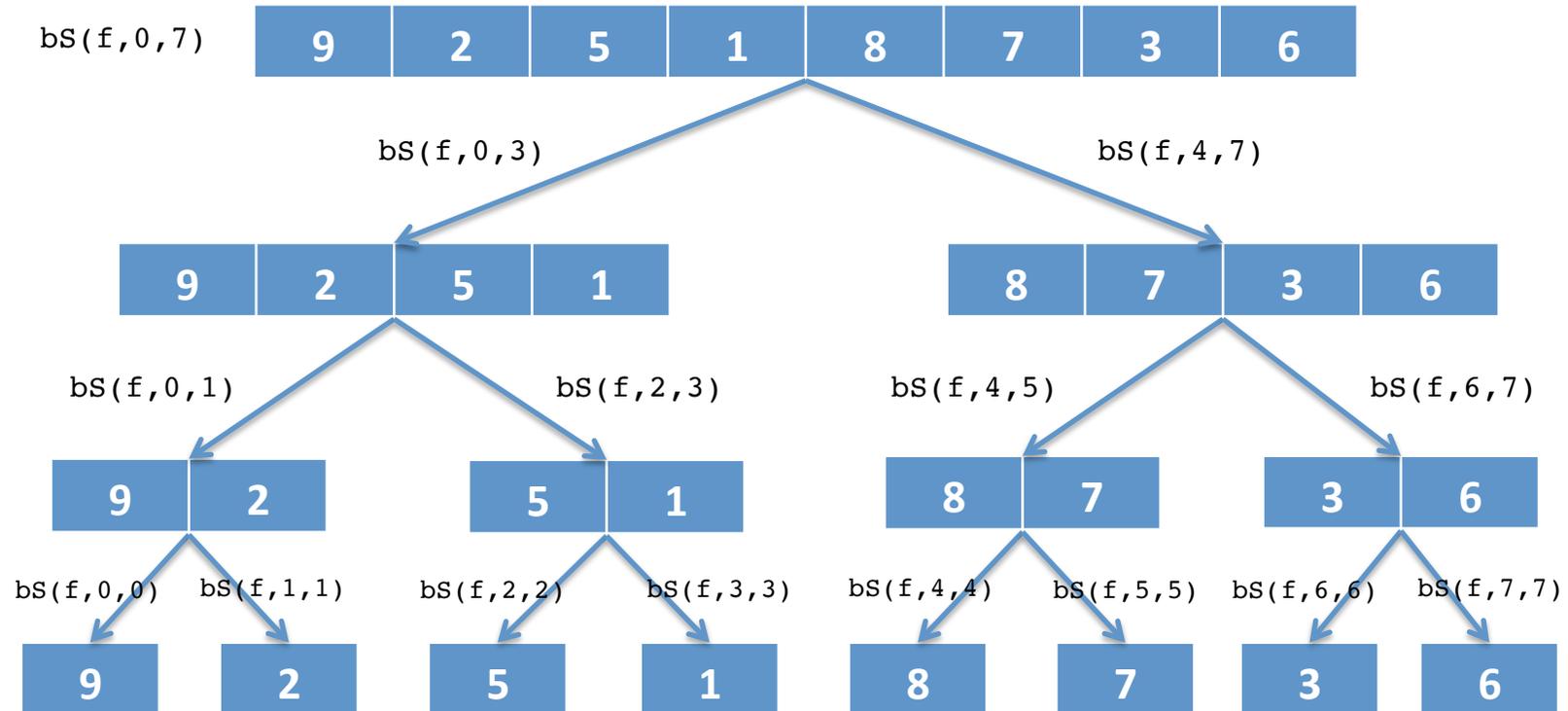
Algorithmus 2/2

```
void merge(int[] list, int u, int o, int d){
    if(2*d>=o-u) compare(list,u,u+d);
    else{
        merge(list, u, o, 2*d);
        merge(list, u+d, o, 2*d);
        for(int i = u+d; i <= o-d; i += 2*d)
            compare(list, i, i+d);
    }
}
```

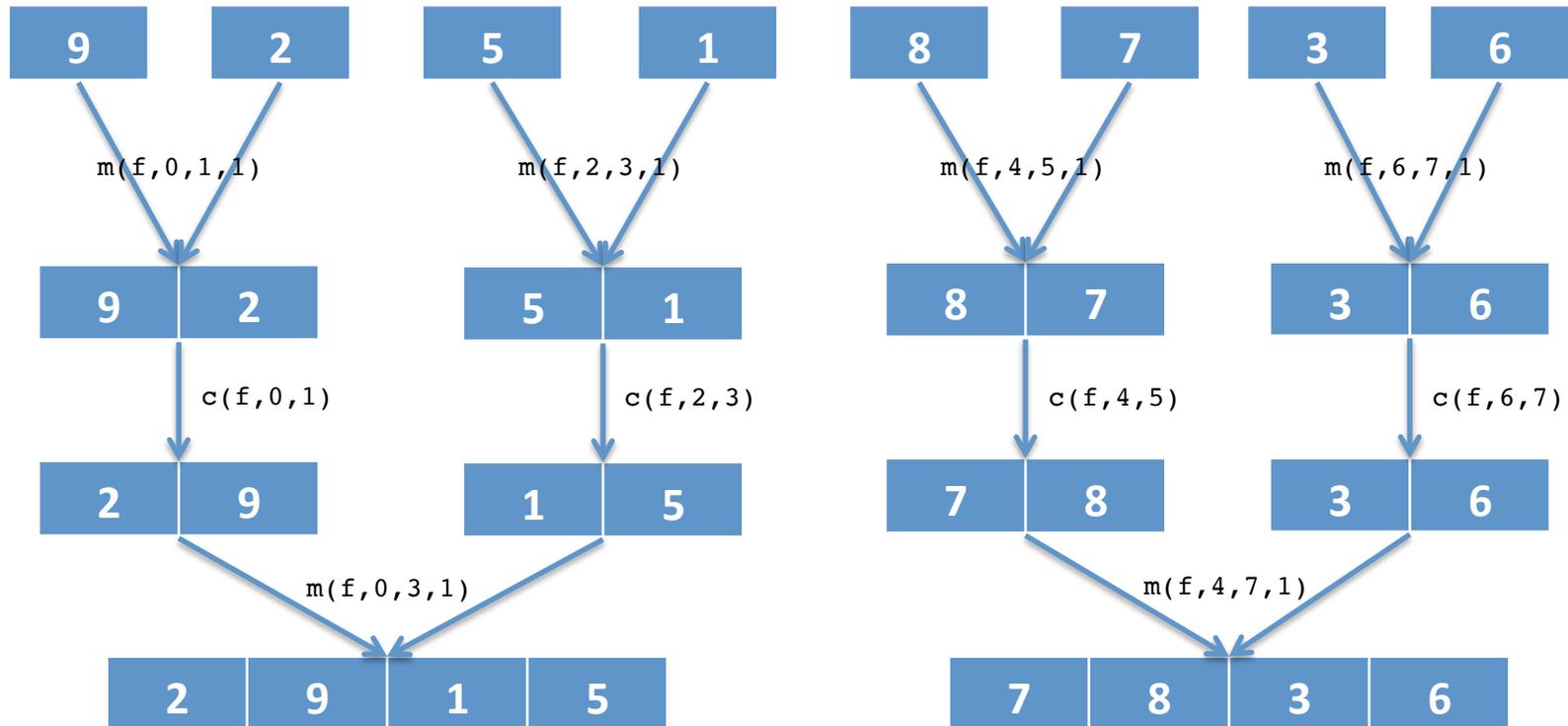
Kann parallel
ausgeführt werden

```
void compare(int[] list, int k, int l){
    if(list[k] > list[l]){
        int b = list[k];
        list[k] = list[l];
        list[l] = b;
    }
}
```

Beispiel 1/5

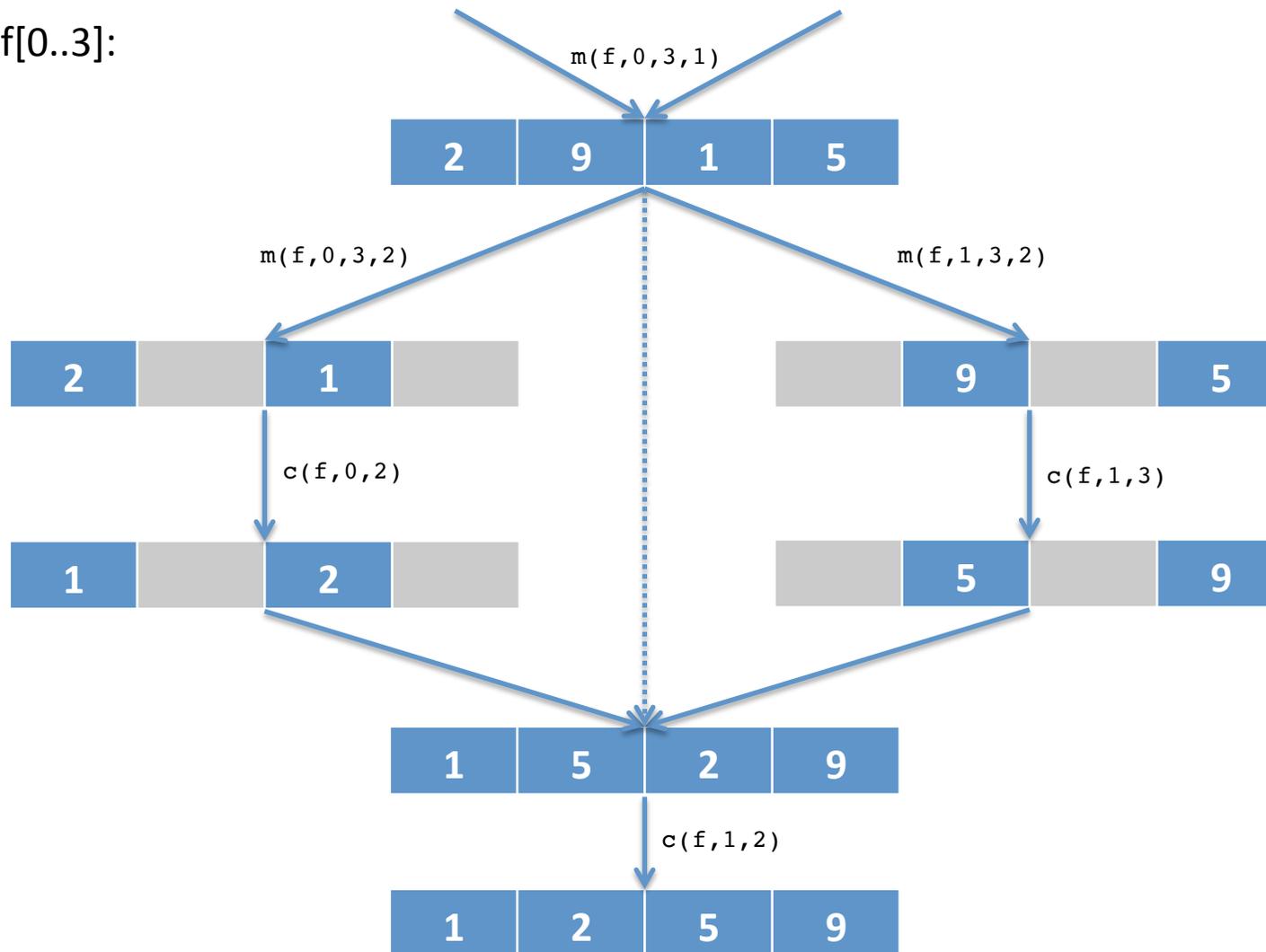


Beispiel 2/5



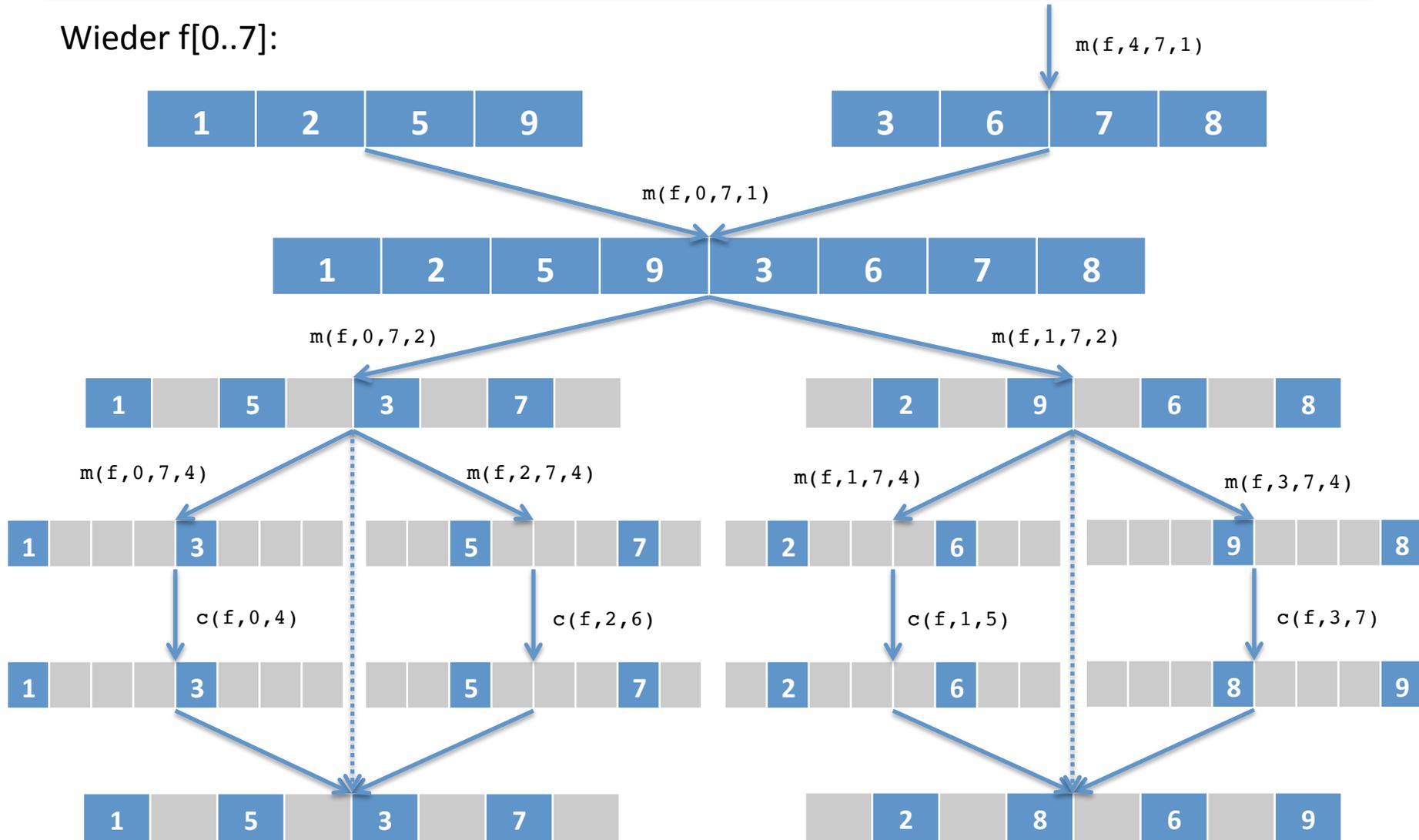
Beispiel 3/5

Nur $f[0..3]$:

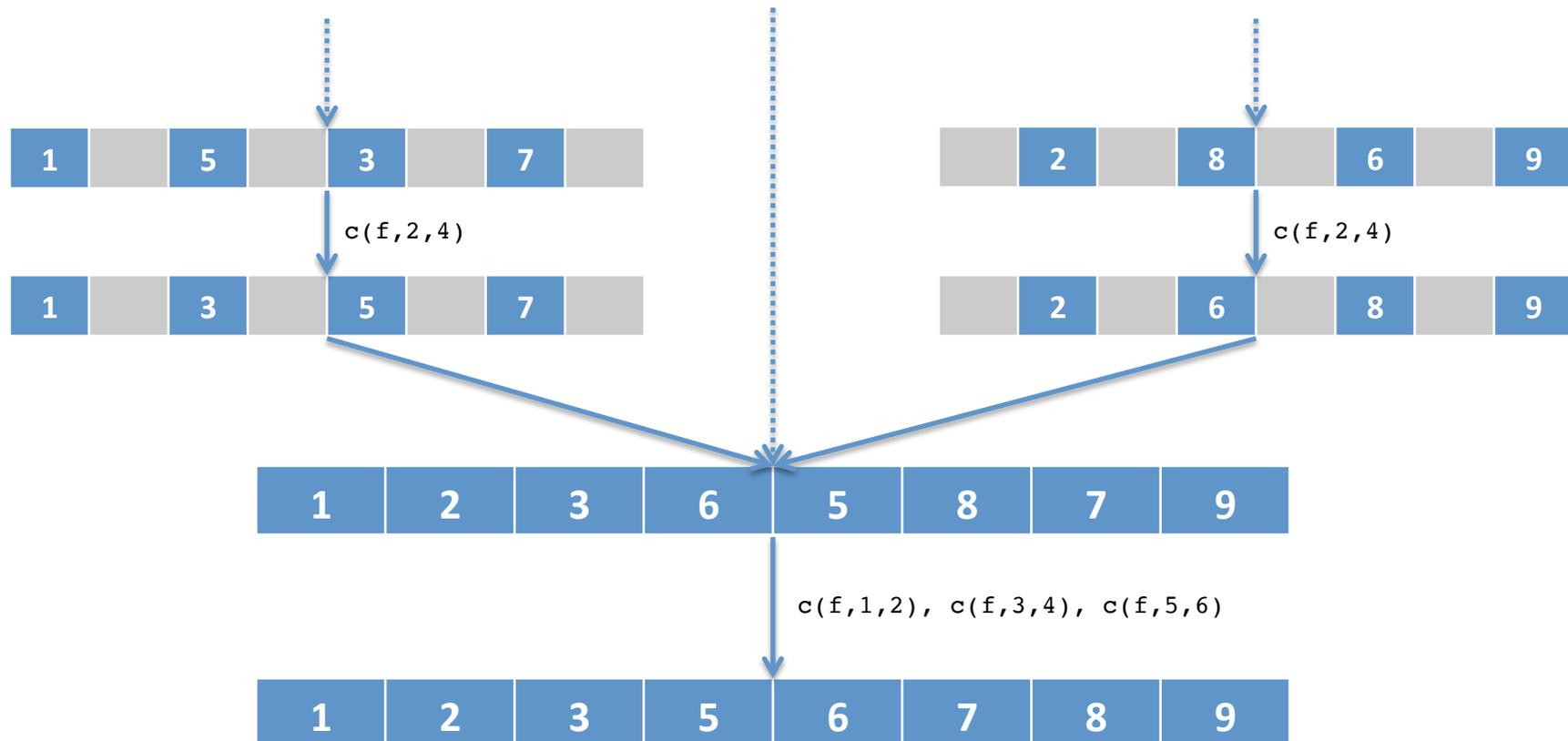


Beispiel 4/5

Wieder $f[0..7]$:



Beispiel 5/5



Analyse

Theorem (Terminierung)

Der Algorithmus Batchersort terminiert für jede Eingabe $\text{int}[]$ f nach endlicher Zeit.

Beweis

Alle rekursiven Aufrufe erhalten stets eine echt kleinere Eingabe und terminieren bei einer einelementigen Eingabe. \square

Analyse

Theorem (Korrektheit)

Der Algorithmus Batchersort sortiert ein Array f korrekt.
ohne Beweis

Theorem (Laufzeit)

Der Algorithmus Batchersort hat bei sequentieller Abarbeitung eine Laufzeit von $O(n \log^2 n)$.
ohne Beweis

Theorem (Laufzeit)

Der Algorithmus Batchersort hat bei paralleler Abarbeitung (Parallelisierung von `batchersort`, `merge`, und `compare`-Aufrufen) eine Laufzeit von $O(\log^2 n)$.
ohne Beweis

Algorithmen und Datenstrukturen

6.2 WEITERE SORTIERPROBLEME

ZUSAMMENFASSUNG

Zusammenfassung

- BucketSort
 - Unter Gleichverteilungsannahmen: $O(n)$
- Batchersort
 - Bei Parallelverarbeitung: $O(\log^2 n)$