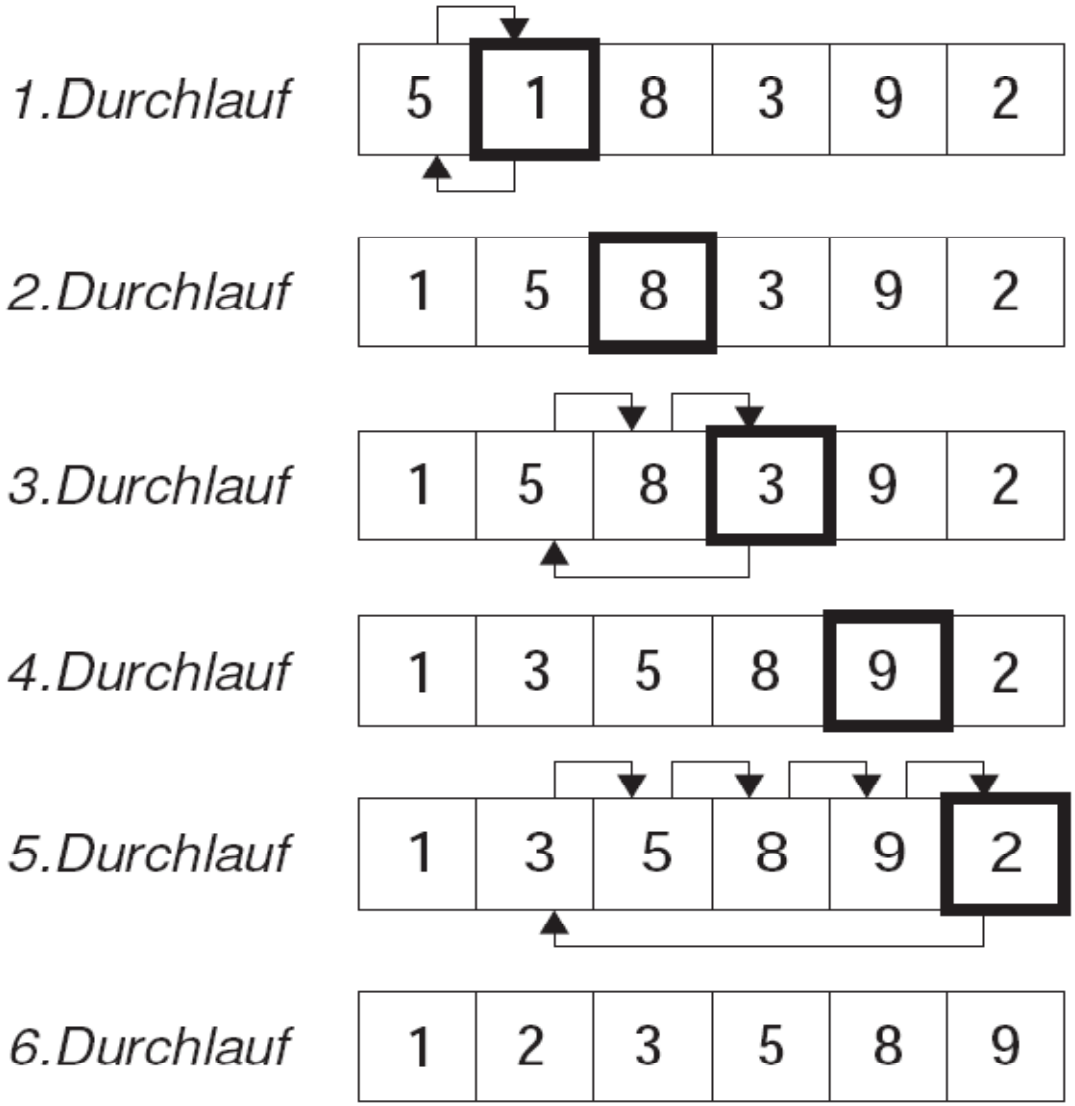
Sortieren durch Einfügen: Prinzip

Idee:

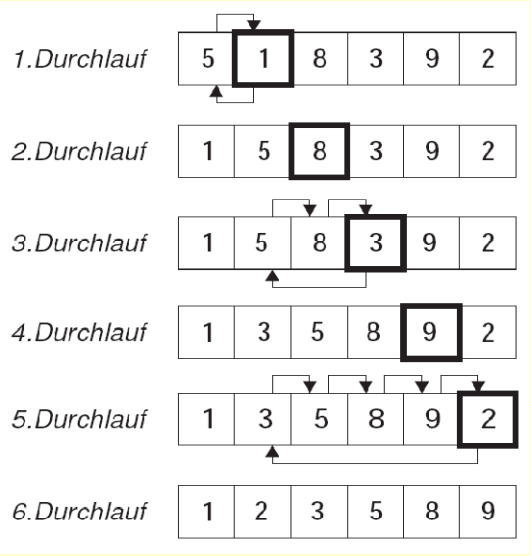
- Umsetzung der typischen menschlichen Vorgehensweise, etwa beim Sortieren eines Stapels von Karten:
  1. Starte mit der ersten Karte einen neuen Stapel
  2. Nimm jeweils nächste Karte des Originalstapels: füge diese an der richtigen Stelle in den neuen Stapel ein
- Im Folgenden: Übertragung dieser Idee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

Sortieren durch Einfügen: Beispiel



Sortieren durch Einfügen: Algorithmus

```
algorithm InsertionSort (F)
Eingabe: zu sortierende Folge F der Länge n
    for i := 2 to n do
        m := F[i]; /* zu merkendes Element */
        j := i;
        while j > 1 do
            if F[j − 1] ≥ m then
                /* Verschiebe F[j − 1] nach rechts */
                F[j] := F[j − 1]; j := j − 1
            else
                Verlasse innere Schleife
            fi;
            F[j] := m /* Einfügeposition */
        od
    od
```



Analyse: InsertionSort

- Aufwand:
  - Anzahl der Vertauschungen
  - Anzahl der Vergleiche
  - Vergleiche **dominieren** Vertauschungen, d.h. es werden (wesentlich) mehr Vergleiche als Vertauschungen benötigt
- Ausserdem Unterscheidung:
  - Bester Fall: Liste ist schon sortiert
  - Mittlerer (zu erwartender) Fall: Liste ist unsortiert
  - Schlechtester Fall: z.B. Liste ist absteigend sortiert

Analyse: InsertionSort (2)

- wir müssen in jedem Fall alle Elemente $i = 2$ bis n durchgehen, d.h. immer Faktor $n - 1$
- außerdem müssen wir die korrekte Einfügeposition finden
- Bester Fall: Liste sortiert
  - Keine Einfügeoperation notwendig (aber Element wird in m zwischengespeichert!)
  - Für jedes Element i ein Vergleich
  - Gesamtanzahl der Vergleiche: $(n - 1) * 1 = n - 1$
  - für Große Listen abgeschätzt als $n - 1 \approx n$
  - ”linearer Aufwand”

Analyse: InsertionSort (3)

- Mittlerer (zu erwartender) Fall: Liste unsortiert (d.h. Einfügeposition wahrscheinlich auf der Hälfte des Rückwegs)
  - bei jedem der $n - 1$ Rückwege Faktor $(i - 1)/2$
  - Gesamtanzahl der Vergleiche:

$$(n-1)/2 + (n-2)/2 + (n-3)/2 + \ldots + 2/2 + 1/2$$

$$= \frac{(n-1) + (n-2) + (n-3) + \ldots + 2 + 1}{2}$$

$$= \frac{1}{2} * \frac{n*(n-1)}{2} = \frac{n*(n-1)}{4}$$

$$\approx \frac{n^2}{4}$$

Analyse: InsertionSort (4)

- Schlechtester Fall: z.B. Liste umgekehrt sortiert (d.h. Einfügeposition am Ende des Rückwegs bei Position 1)
  - bei jedem der $n - 1$ Rückwege Faktor $i - 1$
  - analog zu vorhergehenden Überlegungen, aber doppelte Rückweglänge
  - Gesamtanzahl der Vergleiche: $\frac{n * (n-1)}{2} \approx \frac{n^2}{2}$

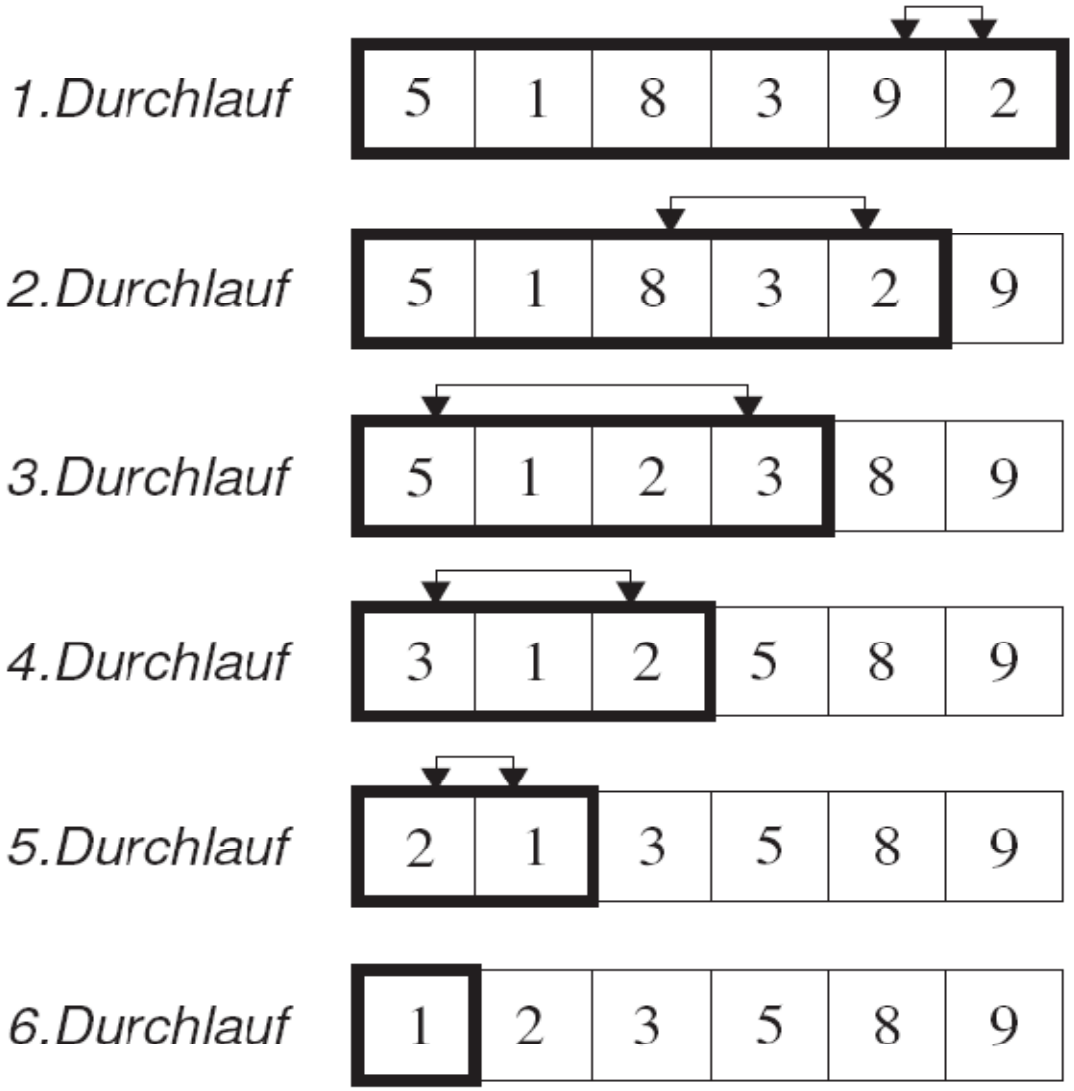
- letzte beiden Fälle: ”quadratischer Aufwand”, wenn konstante Faktoren nicht berücksichtigt werden

Sortieren durch Selektion

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

```
algorithm SelectionSort (F)
Eingabe: zu sortierende Folge F der Länge n
    p := n;
    while p > 0 do
        g := Index des größten Elementes aus F
                im Bereich 1...p;
        Vertausche Werte von F[p] und F[g];
        p := p − 1
    od
```

Sortieren durch Selektion: Beispiel



Analyse: SelectionSort

- in jedem Durchlauf das Element von $F[p]$ mit dem größten Element tauschen
- Variable p läuft von n … 1

  $\Rightarrow$ daher n Vertauschungen
- in jedem Durchlauf das größte Element aus 1 … p ermitteln
- $\Rightarrow p - 1$ Vergleiche

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

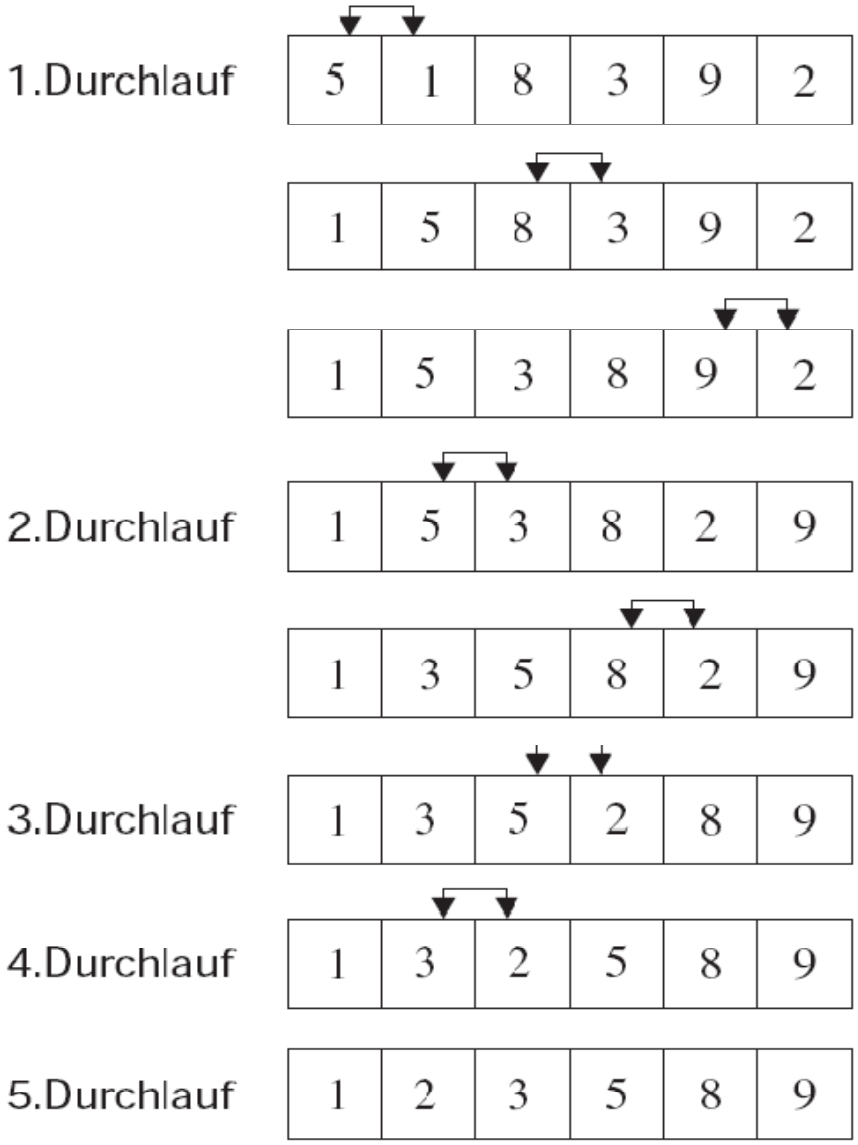
- **Anzahl Vergleiche identisch für besten, mittleren und schlechtesten Fall!**

BubbleSort

- **Idee:** Verschieden große aufsteigende Blasen („Bubbles“) in einer Flüssigkeit sortieren sich quasi von allein, da größere Blasen die kleineren „überholen“.

```
algorithm BubbleSort (F)
Eingabe: zu sortierende Folge F der Länge n
    do
        for i := 1 to n − 1 do
            if F[i] > F[i + 1] then
                Vertausche Werte von F[i] und F[i + 1]
            fi
        od
    until keine Vertauschung mehr aufgetreten
```

BubbleSort: Beispiel



algorithm BubbleSort (*F*)
Eingabe: zu sortierende Folge *F* der Länge *n*
  do
    for *i* := 1 **to** *n* − 1 **do**
      if *F*[*i*] > *F*[*i* + 1] **then**
        Vertausche Werte von *F*[*i*] und F[*i* + 1]
      fi
    od
  until keine Vertauschung mehr aufgetreten

BubbleSort: Optimierung

Beobachtung: Größte Zahl wird in jedem Durchlauf automatisch an das Ende der Liste verschoben

$\Rightarrow$ im Durchlauf j reicht der Vergleich bis Position $n - j$

Analyse: BubbleSort

- Bester Fall: n
- Durchschnittlicher Fall
  - Normal: n^2
  - optimiert: $n^2 / 2$
- Schlechtester Fall
  - normal: n^2
  - optimiert: $n^2 / 2$

Mehr zu Bubblesort:

http://de.wikipedia.org/wiki/Bubblesort

http://www.tcs.ifi.lmu.de/~gruberh/lehre/sorting/sort.html

http://wwwiti.cs.uni-magdeburg.de/iti_db/algoj/code/algoj/kap5/Sort.java

MergeSort: Prinzip

Ursprüngliche Motivation:

- Sortieren von sehr großen Dateien auf externen Medien
  - Aufteilen in kleine Teile, die im Speicher sortierbar sind
  - Zusammenfügen (Mischen/Merge) der Teile nach dem Sortieren in eine vollständig sortierte Datei

Übertragung auf Sortierverfahren im Speicher:

1. Teile die zu sortierende Liste in zwei Teillisten
2. Sortiere diese (rekursives Verfahren!)
3. Mische die Ergebnisse

Zusammenfügen von zwei Folgen (Algorithmus)

procedure Merge (F_1, F_2) $\rightarrow$ F
Eingabe: zwei sortierte Folgen F_1, F_2
Ausgabe: eine sortierte Folge F

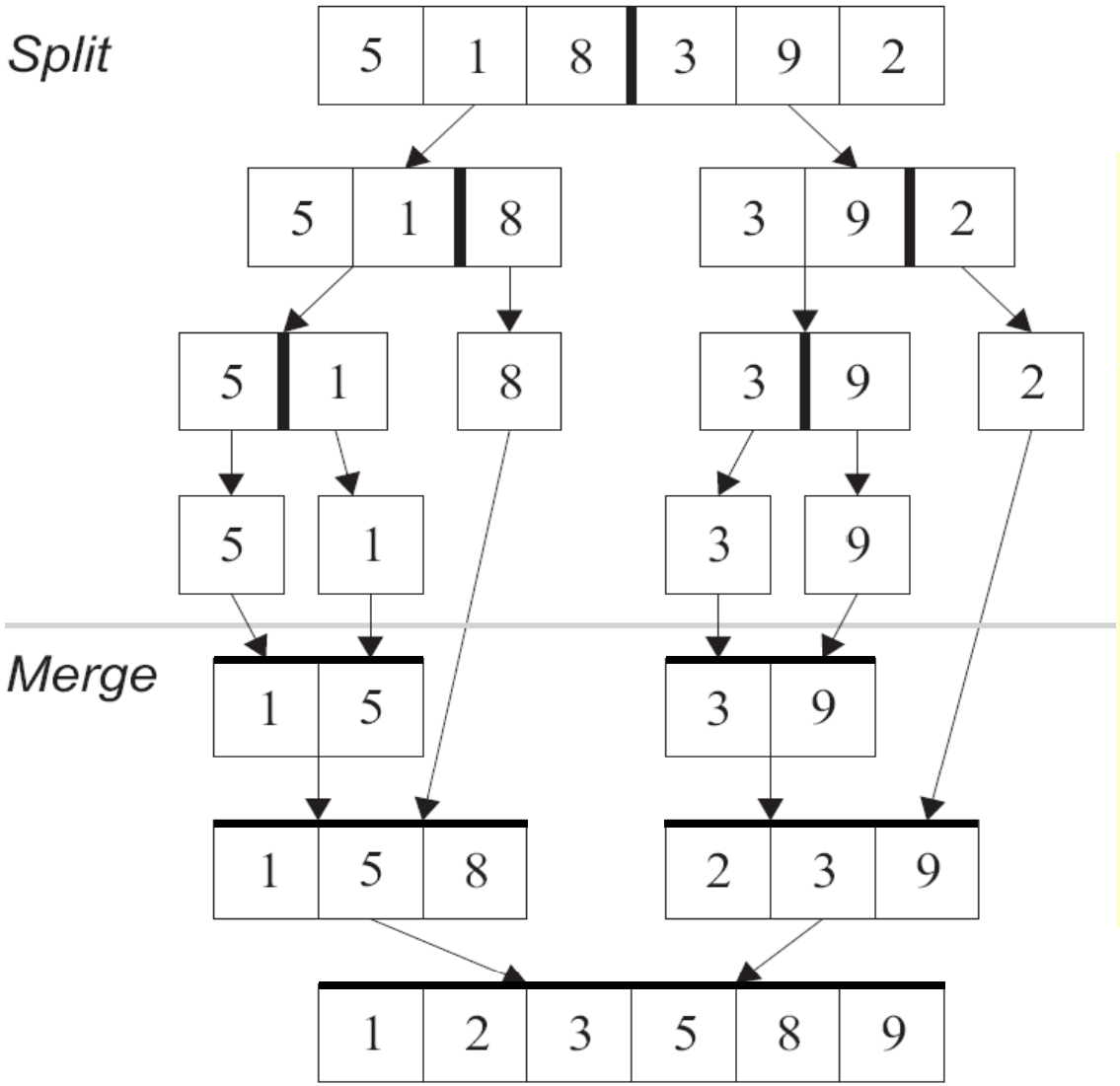
F := leere Folge;
while F_1 oder F_2 nicht leer **do**
    Entferne das kleinere der Anfangselemente
        aus F_1 bzw. F_2;
    Füge dieses Element an F an
od;
Füge die verbliebene nichtleere Folge F_1
    oder F_2 an F an;
return F

Sortieren durch Mischen (Algorithmus)

algorithm MergeSort (F) $\rightarrow F_S$
Eingabe: eine zu sortierende Folge F
Ausgabe: eine sortierte Folge F_S

```
if F einelementig then
   return F
else
   Teile F in F1 und F2;
   F1 := MergeSort (F1);
   F2 := MergeSort (F2);
   return Merge (F1, F2)
fi
```

MergeSort: Beispiel



algorithm
MergeSort (F) → FS
Eingabe:
eine zu sortierende Folge F
Ausgabe:
eine sortierte Folge FS

if F einelementig **then**
return F
else
Teile F in F_1 und F_2;
F_1 := MergeSort (F_1);
F_2 := MergeSort (F_2);
return Merge (F_1, F_2)
fi

Analyse: MergeSort

- Für Anzahl der Vergleiche V_n für Folge der Länge n gilt:
  - jede Teilfolge sortieren: $V_{n/2}$ Vergleiche (d.h. Anzahl der Vergleiche für das Sortieren einer Folge halber Länge)
  - Mischen: n Vergleiche
  - Somit insgesamt:

$$V_n = 2V_{n/2} + n \quad \text{für} \quad n \geq 2 \text{ mit } V_1 = 0$$

- Annahme: $n = 2^N$ (n ist Zweierpotenz):

$$V_{2^N} = 2V_{2^{N-1}} + 2^N \quad \Rightarrow \quad \frac{V_{2^N}}{2^N} = \frac{V_{2^{N-1}}}{2^{N-1}} + 1$$

Analyse: MergeSort (2)

- Sukzessive $\frac{V_{2^{N-i}}}{2^{N-i}}$ durch $\frac{V_{2^{N-i-1}}}{2^{N-i-1}}+1$ ersetzen bis $i = N$:

$$\begin{aligned}\frac{V_{2^N}}{2^N} &= \frac{V_{2^{N-2}}}{2^{N-2}}+1+1\\ &= \frac{V_{2^0}}{2^0}+\cdots+1\\ &= N\end{aligned}$$

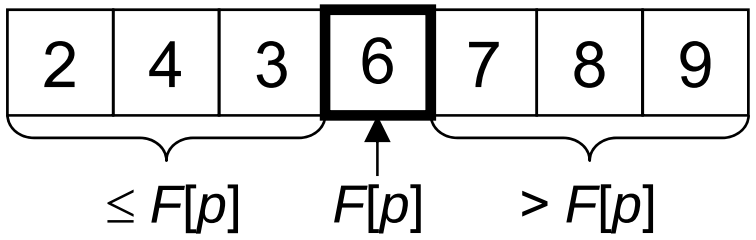
- da aus $n = 2^N$ folgt $N = \log_2 n$ ergibt sich für V_n:

$$\begin{aligned}\frac{V_n}{n} &= \log_2 n\\ V_n &= n\log_2 n\end{aligned}$$

QuickSort: Prinzip

Idee:

- ähnlich wie MergeSort durch rekursive Aufteilung
- Vermeidung des Mischvorgangs durch Aufteilung der Teillisten in zwei Hälften bezüglich eines Pivot-Elementes,
- Wobei
  - in einer Liste alle Elemente größer als das Referenzelement sind
  - in der anderen Liste alle Elemente kleiner sind



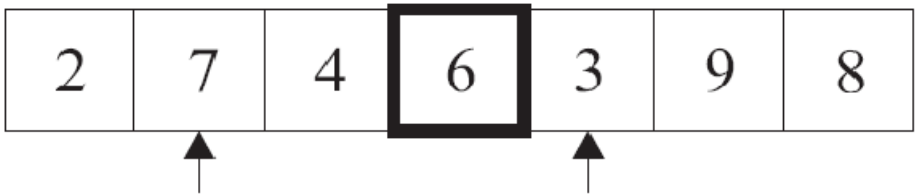
QuickSort: Algorithmus

`algorithm` QuickSort (**F**, u, o)
Eingabe: eine zu sortierende Folge **F**, die untere und obere Grenze u, o

```
if o > u then
    Zerlege F[u...o] in zwei Partitionen
            F[u...r] und F[l...o];
    QuickSort (F, u, r);
    QuickSort (F, l, o);
fi
```

QuickSort: Vertauschen von Elementen

- **Pivot-Element** *p*
  - Folge von links durchsuchen, bis Element gefunden, das größer oder gleich *p* ist
  - Folge von rechts durchsuchen, bis Element gefunden, das kleiner oder gleich *p* ist
- Elemente ggf. tauschen

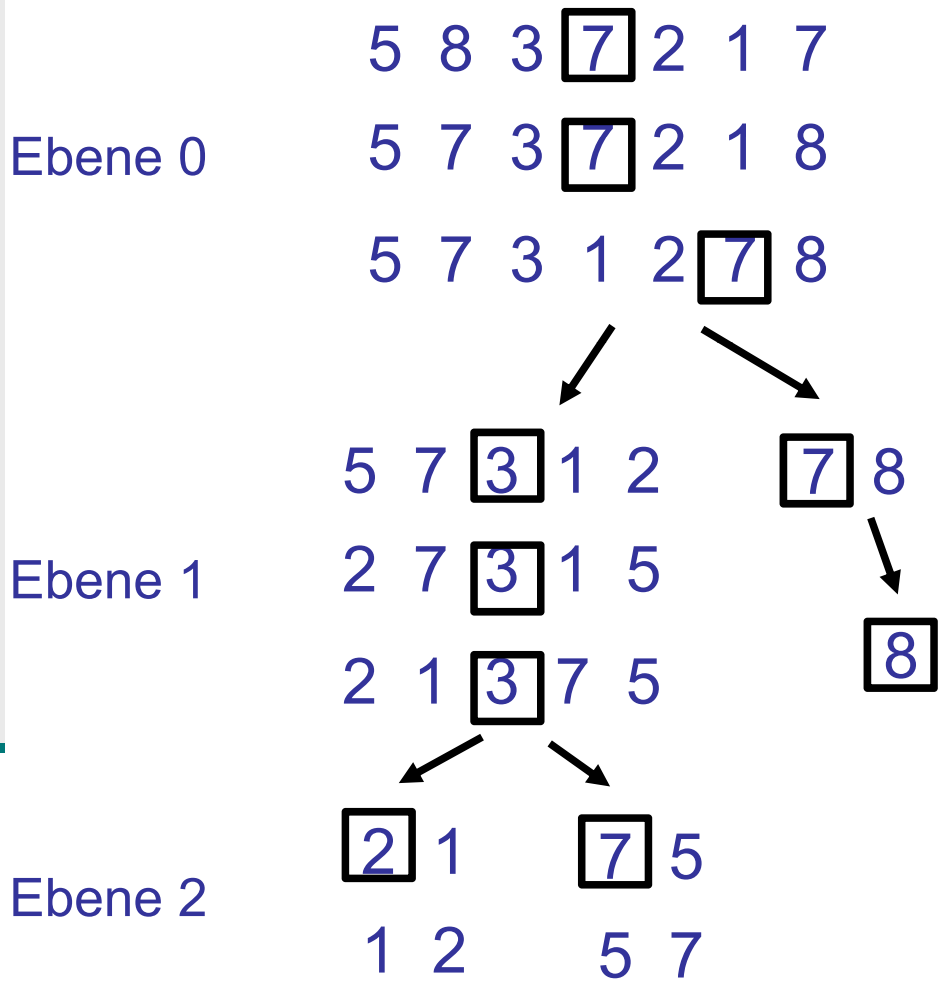


QuickSort: Zerlegen beim QuickSort

```
algorithm Zerlege (F, u, o)
Eingabe:  Folge F, untere/obere Grenze u, o,
Ausgabe:Positionen l und r der Zerlegung

    p := F[(u+o)/2];
    l := u;   r := o;
    while l <= r do
        l := Index des ersten Elementes aus l...o
                mit F[l] >= p
        r := Indes des ersten Elementes aus u...r
                mit F[r] <= p
        if l <= r then
            Tausche F[l] und F[r];
            l := l+1; r := r-1;
        fi
    od
```

QuickSort: Beispiel




```
algorithm QuickSort (F, u, o)
Eingabe: eine zu sortierende Folge F,
die untere und obere Grenze u, o

if o > u then
Zerlege F[u...o] in zwei Partitionen
                F[u...r] und F[l...o];
QuickSort (F, u, r);
QuickSort (F, l, o);
fi
```

```
algorithm Zerlege (F, u, o)
Eingabe: F, untere/obere Grenze u, o,
Ausgabe: Positionen l, r der Zerlegung
p := F[(u+o)/2];
l := u;   r := o;
while l <= r do
  l := Index des ersten Elementes
       aus l...o mit F[l] >= p
  r := Indes des ersten Elementes
       aus u...r mit F[r] <= p
  if l <= r then
     Tausche F[l] und F[r];
     l := l+1; r := r-1;
  fi
od
```

QuickSort in Java

```
void qsort(int[] F, int u, int o){
  int l = u, r = o;
  if (o > u){                        // if (r > l){
     int p = F[(u+o)/2];
     while (l <= r){
         while (l < o && F[l] < p) ++l;
         while (r > u && F[r] > p) --r;
         if (l <= r){
            swap(F, l, r);
            ++l; --r;
         }
     }
     if (u < r) qsort(F, u, r);
     if (l < o) qsort(F, l, o);
  }
}
static void quickSort (int[] array) {
   qsort (array, 0, array.length - 1);
}
```

Analyse: QuickSort

- Aufwand analog zu MergeSort abschätzbar
  - Bester Fall: $n \log_2 n$
  - Durchschnittlicher Fall: $1.38\, n \log_2 n$
  - Schlechtester Fall: $n^2 \log_2 n$
- aber:
  - QuickSort benötigt weniger Ressourcen
  - im Gegensatz zur MergeSort ist QuickSort durch Vorgehensweise bei Vertauschungen **instabil**

Sortierverfahren im Vergleich

Verfahren	Stabilität	Vergleiche (im Mittel)
SelectionSort	instabil	$\approx n^2 / 2$
InsertionSort	stabil	$\approx n^2 / 4$
BubbleSort	stabil	$\approx n^2 / 2$
MergeSort	stabil	$\approx n \log_2 n$
QuickSort	instabil	$\approx 1.38n \log_2 n$

Zusammenfassung

- Suchen und Sortieren als Grundaufgaben in der Informatik
- Grundalgorithmen
- Algorithmenmuster „Teile und herrsche“
- Abschätzung des Aufwands
- Literatur: Saake/Sattler: Algorithmen und Datenstrukturen Kapitel 5
- Beispielcode in Java:

http://wwwiti.cs.uni-magdeburg.de/iti_db/algoj/code.html