Algorithmen und Datenstrukturen

Suchen und Sortieren

Algorithmen auf Sequenzen:

Suchen in Sequenzen

(*Hier:* Beschränkung auf Sequenzen von Zahlen)

Lineare Suche

Name: **Lineare Suche in Sequenz (von Zahlen)**
Eingabe: Sequenz L von n Zahlen, Zahl k
Ausgabe: eine Position, an der k in L auftritt,
-1 falls k nicht in L enthalten ist

```
# findet erste Position des Auftretens, falls k in L
def linSearch(L,k):
    for i in range(0, len(L)):
        if (L[i] == k): return i
    return -1
```

→ $O(n)$ Laufzeit

Laufzeitanalyse

- **worst case**
  k nicht in *L* → **θ(*n*)**

- **best case**
  k an erster Position → **θ(1)**

- **average case** (*Annahme:* alle Fälle gleich wahrscheinlich)
  n+1 Fälle: 1 Vergleich, 2 Vergleiche, ..., *n* Vergleiche, *n* Vergleiche

$$\frac{1}{n+1}\left(\sum_{i=1}^{n} i + n\right) = \frac{n(n+1)}{2(n+1)} + \frac{n}{n+1} \in \boldsymbol{\theta(n)}$$

Binäre Suche

- falls L sortiert vorliegt

1. Untersuchung des Elements $L(m)$ in der Mitte von L
2. Falls Treffer $\rightarrow$ gib Index m zurück

   Sonst:

3. Suche in der passenden Hälfte weiter
   a) falls $L[m] > k$: suche in linker Hälfte weiter
   b) falls $L[m] < k$: suche in rechter Hälfte weiter

   (*Rekursion*)

Binäre Suche

```
# binaere Suche nach k in Teilfolge von i bis j
# pre: L ist nicht leer und aufsteigend sortiert
# pre: 0 <= i <= j <= len(L)-1
def binSearchT(L,k,i,j):
      m = (i + j) // 2
      if (L[m] == k): return m
      if (L[m] > k and m > i):
            return binSearchT(L,k,i,m-1)
      if (L[m] < k and m < j):
            return binSearchT(L,k,m+1,j)
      else: return -1
```

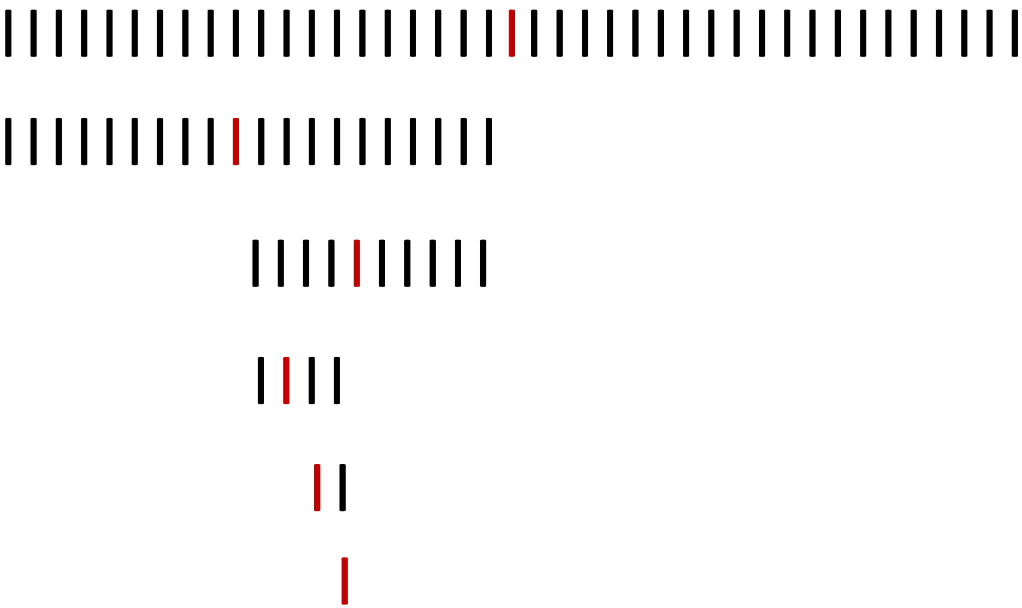
Binäre Suche

```
# binaere Suche nach k in L
# pre: L ist aufsteigend sortiert

def binSearch(L,k):
      if (len(L) == 0):
            return -1
      else:
            return binSearchT(L,k,0,len(L)-1)
```

Laufzeitanalyse Binäre Suche

- fortgesetztes Halbieren der Folge, die noch durchsucht werden muss



maximal
$1+\lfloor \log_2 n \rfloor$
viele
Aufrufe

$\rightarrow O(\log n)$

Laufzeitanalyse Binäre Suche (2)

- $O(\log n)$ viele Aufrufe
- je Aufruf: $O(1)$ viel Aufwand:

$\rightarrow$ **$O(\log n)$**

```
m = (i + j) // 2
if (L[m] == k): return m
if (L[m] > k and m > i):
      return binSearchT(L,k,i,m-1)
if (L[m] < k and m < j):
      return binSearchT(L,k,m+1,j)
else: return -1
```

Algorithmen auf Sequenzen:

Sortieren von Sequenzen

(*Hier:* Beschränkung auf Sequenzen von Zahlen)

Sortieren (*naiv*)

Nacheinander die Liste „korrekt besetzen“:

für alle Indexwerte *i*

Vergleich aller Werte rechts von *i* mit dem Wert an Position *i*

Wenn kleinerer Wert gefunden, dann mit diesem weiter vergleichen

Wert an Position *i* mit kleinstem Wert rechts von *i* vertauschen

```
# Vertauschen der Werte an Positionen i und j in Liste L
# pre: 0 <= i, j <= len(L)-1
def chg(L,i,j):
      tmp = L[i]
      L[i] = L[j]
      L[j] = tmp
```

$O(1)$

Sortieren (*naiv*)

Name: **SelectionSort**
Eingabe: Sequenz L von n Zahlen
Ausgabe: keine, sortiert die Elemente von L aufsteigend

```
def selectionSort(L):
    for i in range(0,len(L)):
        indexOfMin = i
        for j in range(i+1,len(L)):
            if L[indexOfMin] > L[j]:
                indexOfMin = j
        chg(L,i,indexOfMin)
```

$\Theta(n^2)$ (best case = worst case = average case)

Sortieren durch Einfügen in neue Liste

- ausgehend von leerer Liste die Liste elementweise aufbauen, wobei das nächste Element immer an der „richtigen Position" eingefügt wird (*„Zwischenergebnis" ist stets sortiert*)

- benutzt Hilfsfunktion zum sortierten Einfügen eines Elements in eine sortierte Liste

Einfügen

Name: Insert
Eingabe: *sortierte* Sequenz *L* von *n* Zahlen, Zahl *k*
Ausgabe: sortierte Sequenz von *n*+1 Zahlen,
die neben den Elementen von *L* zusätzlich *k* enthält

```
def insert(L,k):
    L = L + [k]      # Liste verlängern
    i = len(L)-1
    # groessere Elemente nach rechts
    while ((i > 0) and (L[i-1] > k)):
        L[i] = L[i-1]
        i = i - 1
    if (i != len(L)-1):
        L[i] = k     # an richtiger Stelle einfügen
    return L
```

Analyse insert

```
def insert(L,k):
    L = L + [k]
    i = len(L)-1
    while ((i > 0) and (L[i-1] > k)):
        L[i] = L[i-1]
        i = i - 1
    if (i != len(L)-1):
        L[i] = k
    return L
```

- **best case:** **θ(1)**
- **worst case:** **θ(*n*)**
- **average case:** **θ(*n*)** $\approx \frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2\,n} = \frac{n+1}{2}$

Sortieren durch Einfügen in neue Liste

```
def insertionSort_new(L):
    sortList = []
    for i in range(0,len(L)):
        sortList = insert(sortList, L[i])
    return sortList
```

- Laufzeit: $\theta(n \cdot t_{\text{insert}}(n))$, also $\theta(n^2)$
- Platzbedarf: $2 \cdot n + c$ (für eine Konstante $c \geq 0$)
  - neben `L` muss `sortlist` aufgebaut werden
  - im Hauptprogramm muss Zuweisung des Rückgabewertes erfolgen (Einheitlichkeit??!!)

In-place Algorithmen

- Prozeduren, die die Eingabedaten im Speicher direkt manipulieren
  - keine Vervielfältigung des Speicherbedarfs
  - Nebeneffekt an Eingabedaten

- Beispiel: SelectionSort

InsertionSort als in-place Algorithmus

- Aufbau der sortierten Liste in *L* von links nach rechts
- Einfügen des nächsten Elements *val* von Position *pos* in bereits sortierte Teilfolge links von *pos* (für *pos* $\geq$ 1)
  - Vergleich von *val* mit Werten ab *pos*-1 von rechts nach links
  - entweder größerer Wert gefunden → nach rechts schieben
  - oder *val* an aktuell „freier" Stelle einfügen

≤	≤	≤	>	>	***val***				

pos

- bereits sortierte Teilfolge -

≤	≤	≤	***val***	>	>				

$\boxed{\Theta(n^2)}$

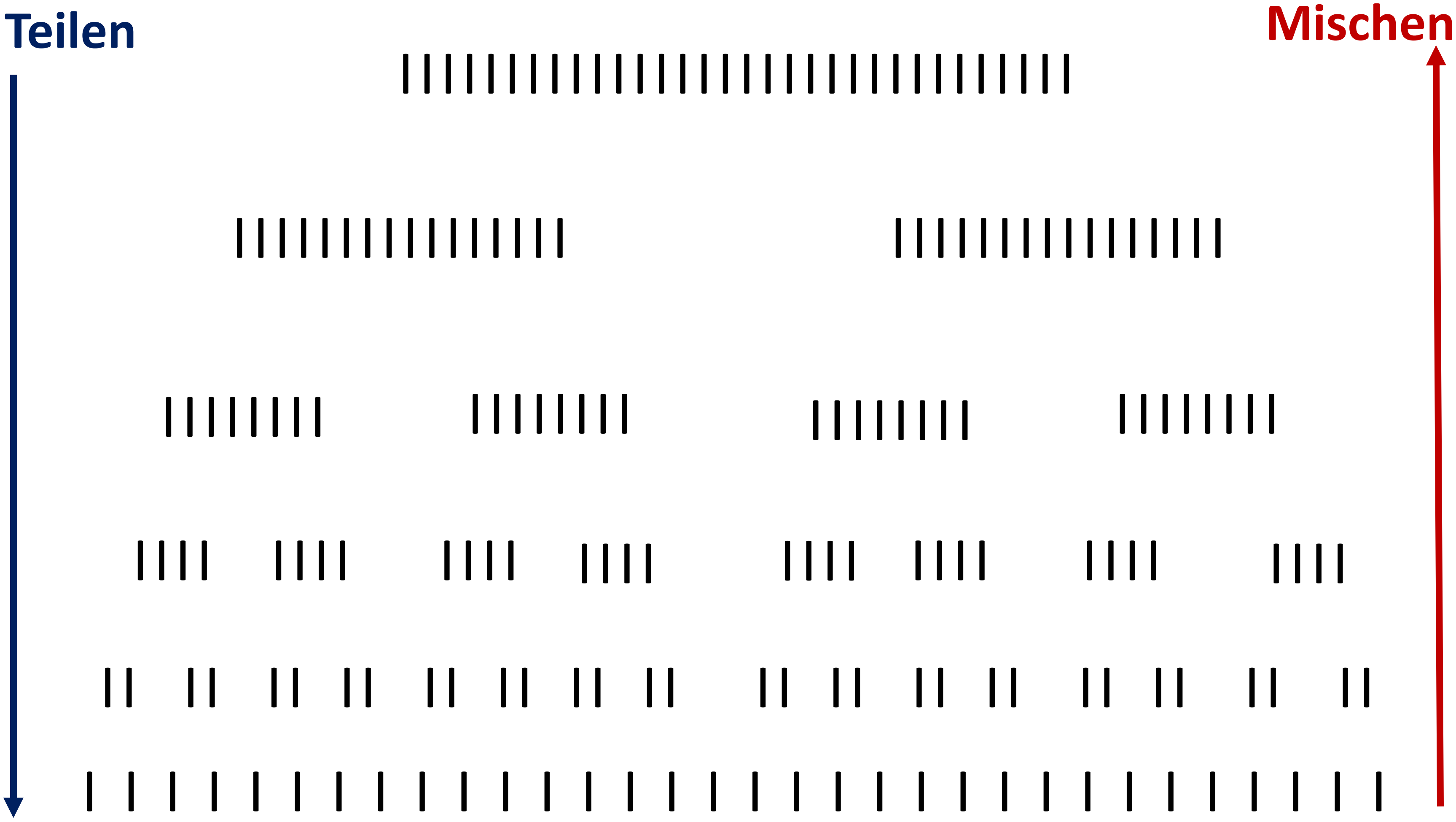
InsertionSort als in-place Algorithmus

```
def insertionSort(L):
    for i in range(1,len(L)):
        val = L[i]
        pos = i
        # durchsuche L rueckwaerts nach korrekter
        # Position für val
        while ((pos > 0) and (L[pos-1] > val)):
            L[pos] = L[pos-1]
            pos = pos-1
        if pos != i:
            L[pos] = val
```

Teile und Herrsche beim Sortieren

- kein Vorteil bei SelectionSort oder InsertionSort
- **Mergesort**
  Sortieren durch Mischen bereits sortierter Teilfolgen
  1. Teilen der Sequenz in möglichst gleich große Teilfolgen
  2. Sortieren der Teilfolgen (*Rekursion*)
     → Sortieren von Sequenzen der Länge 1 ist trivial
  3. Zusammenfügen durch Mischen
     (Funktion **`merge(`**S_1**`,`**S_2**`)`** wie in GdP, 2. Übung)
     → Tatsache, dass Teilfolgen sortiert sind, wird ausgenutzt
     → lineare Laufzeit für das Zusammensetzen der Lösung

Mergesort



Mergesort Beispiel: Teilen

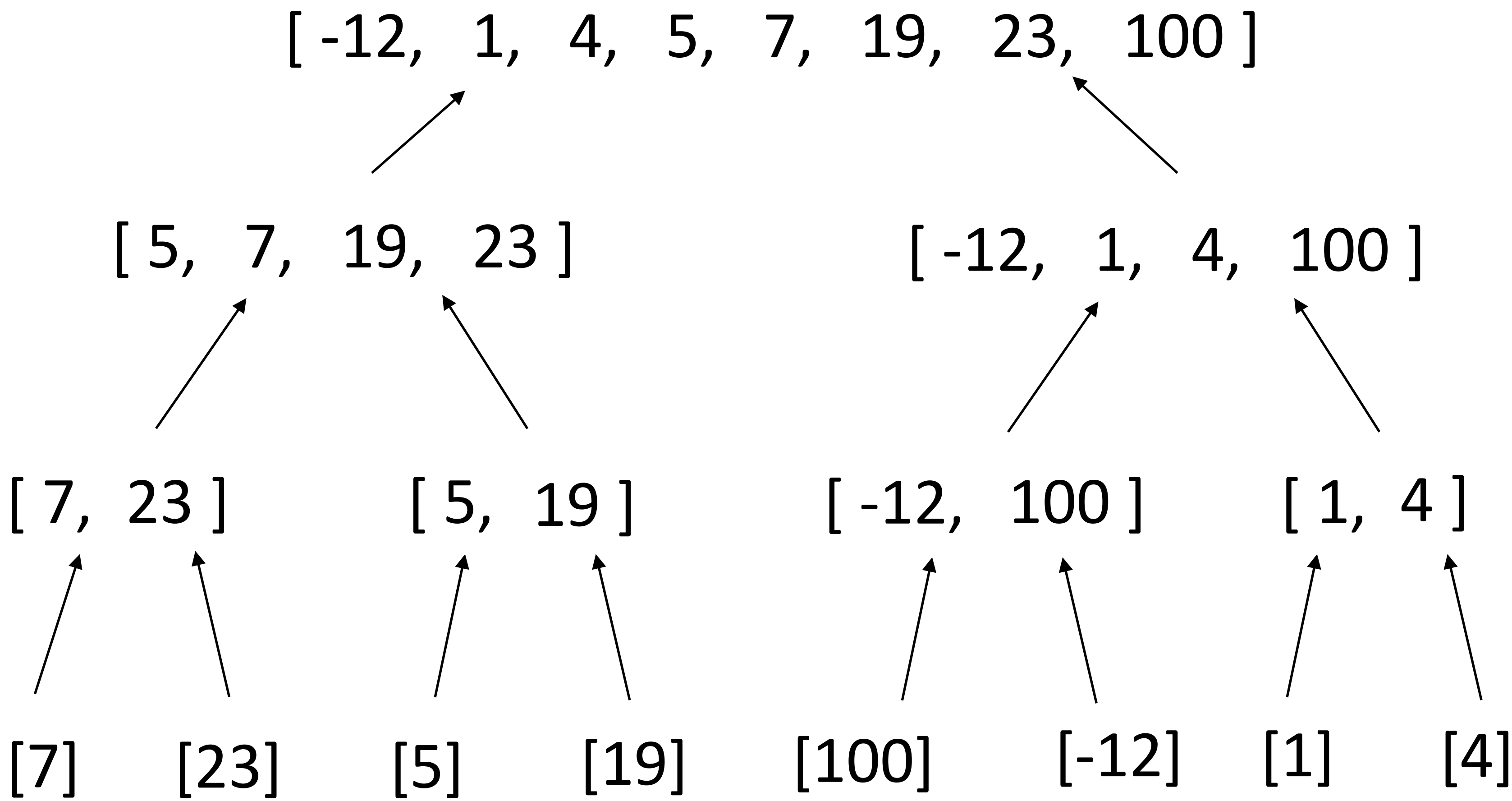
[7, 23, 5, 19, 100, -12, 1, 4]

[7, 23, 5, 19]   [100, -12, 1, 4]

[7, 23]   [5, 19]   [100, -12]   [1, 4]

[7]   [23]   [5]   [19]   [100]   [-12]   [1]   [4]

Mergesort Beispiel: Mischen



Mergesort: Laufzeitanalyse

- best = average = worst case:
- *intuitiv:* logarithmisch viele Ebenen; je Ebene:
  - **θ(1)** zum Test, ob Länge 1, Berechnung der Mitte, rekursive Aufrufe
  - **θ(*n*)** zum Mischen
  - ➢ Gesamtaufwand: **θ(*n* log *n*)**
- *Mit Master-Theorem:*
  - $t(n) = 2t(n/2) + c \cdot n \quad (a = b = 2, \;\; \alpha = 1 \;\rightarrow\; a = b^{\alpha})$
  - **θ(*n* log *n*)**
- ist kein in-place Algorithmus

Sortieren durch Teilen ohne Mischen

- **Quicksort** (Sir Antony (Tony) Hoare, 1961)
- Auswahl eines beliebigen Elements $\boldsymbol{s} = L[p]$ (*Pivot-Element*)
- Bilden der Teilfolge $\boldsymbol{T_1}$ aller Elemente $L[i] < s$
  und der Teilfolge $\boldsymbol{T_2}$ aller Elemente $L[i] \geq s$
- Rekursiver Aufruf von Quicksort für beide Teilfolgen
- *Zusammengesetzte Lösung:*
  1. Sortierte Teilfolge $\boldsymbol{T_1}$, gefolgt von
  2. sortierter Teilfolge $\boldsymbol{T_2}$

Quicksort (Pseudocode)

- **quickSort**(*L*, *low*, *high*) für eine Teilfolge von *L* von *low* bis *high*
- $0 \leq low \leq high \leq |L|$
- Hilfsfunktion **teile**(*L*, *low*, *high*):
  - bestimmt Pivot-Element *s*
  - Verschieben der Elemente von *L* so, dass ab *low* Teilfolge T_1 aller Elemente kleiner *s* stehen, gefolgt von Teilfolge T_2 aller anderen Elemente
  - berechnet Position *j*, an der T_1 endet
  - berechnet Position *i*, von der an die Teilfolge T_2 zu sortieren ist

```
teile(L, low, high)     # liefert j und i
falls low < j
        quickSort(L, low, j)
falls i < high
        quicksort(L, i, high)
```

Für festes low, high:

$$t(n) \in \Theta(t_{\text{teile}}(n))$$

Funktion teile(*L*, *low*, *high*)

p berechnen (z.B. $p = (low + high)//2$)

$s \leftarrow L[p]$

$i \leftarrow low$, $j \leftarrow high$

führe aus

solange $L[i] < s$

$i \leftarrow i + 1$

solange $L[j] > s$

$j \leftarrow j - 1$

falls $i \leq j$

vertausche $L[i]$ mit $L[j]$ in der Sequenz L

$i \leftarrow i + 1$

$j \leftarrow j - 1$

bis $i > j$ erreicht ist

j ist Position, an der T_1 endet, ab i muss T_2 sortiert werden

$$\boxed{t_{teile}(n) \in \Theta(n)}$$

Funktion teile(*L*, *low*, *high*) – Beispiel

low							high
7	23	5	19	100	-12	1	4
i							*j*
7	4	5	19	100	-12	1	23
		i				*j*	
7	4	5	1	100	-12	19	23
				i	*j*		
7	4	5	1	-12	100	19	23
				j	*i*		

Quicksort-Algorithmus

```
quicksort(L, low, high)
    teile(L, low, high)     # liefert j und i
    falls low < j
        quickSort(L, low, j)
    falls i < high
        quicksort(L, i, high)
```

Effizienz Quicksort

- **ist in-place Algorithmus**
- **worst case**
  - Pivot-Element ist kleinstes oder größtes Element
  - Rekursionstiefe $\theta(n)$ (nächste Liste ist immer nur um 1 kleiner)
  - Laufzeit $\boldsymbol{\theta(n^2)}$
- **best case**
  - Listen werden nahezu halbiert
  - Rekursionstiefe $\theta(\mathit{log}\ \mathrm{n})$
  - Laufzeit $\boldsymbol{\theta(n \log n)}$
- **average case**
  - man rechnet aus: Länge der längeren Teilliste nach dem Teilen: $\frac{3}{4}n - \frac{3}{2}$
  - Laufzeit $\boldsymbol{\theta(n \log n)}$

Vergleich von Mergesort und Quicksort

- Mergesort besser im worst case.
  - Mergesort, wenn sehr viele verschiedene Folgen sortiert werden müssen.
- In der Praxis ist Quicksort oft überlegen.
  - recht geringe Wahrscheinlichkeit für den worst case
  - Gute Auswahl des Pivot-Elements! *Besonders vorteilhaft*: zufällige Position oder Median dreier Werte
  - Vergleich mit immer demselben Element (Pivot-Element)
  - in der Praxis: relativ wenige Vertausch-Operationen
  - Mergesort muss viele Daten kopieren (Hilfssequenz) → *nicht in-place*