Algorithmen und Datenstrukturen

Kapitel 03: Sortieren

Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Sortieren: Motivation

Warum ist Sortieren wichtig?

1. Praxisrelevanz

- Sortieren ist praxisrelevant: **25% aller Rechenzeit** entfällt auf Sortiervorgänge[1].
- **Beispiele**: Datenbanken, Suchmaschinen, ...

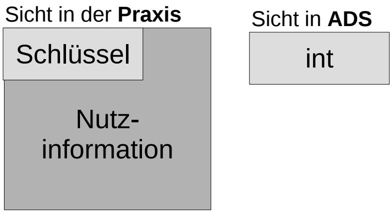
2. Didaktik

Sortieren ist **das Einführungsproblem** der Algorithmik.

- Es gibt eine Vielfalt an Ansätzen.
- Viele Standardkonzepte werden behandelt *(O-Notation, Rekursion, Divide-and-Conquer, untere Schranken, ...)*

[1]Schätzung aus den 1960er Jahren... Trotzdem auch noch heute wichtig!

Sortieren: Formalisierung



Praxis

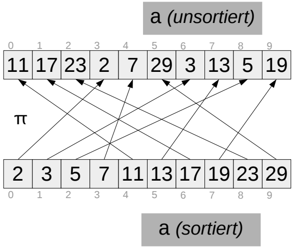
- In der Praxis sortieren wir **beliebige Objekte** *(z.B. Kunden, Dokumente, ...)*.
- Diese Objekte enthalten jeweils einen **Schlüssel** *(z.B. Kundennummer)* und **Nutzinformation** *(z.B. Umsatz, Name, Adresse, ...)*.
- Der **Schlüssel** definiert eine **Ordnungsrelation** auf Objekten.

ADS: Reduktion aufs Wesentliche

Was wir sortieren ist eigentlich irrelevant: Wir benötigen nur Objekte mit einer gegebenen **Ordnungsrelation**. Wir vereinfachen:

- Die Objekte sind `int`-Zahlen.
- die Ordnungsrelation ist $\leq$ / $\geq$.
- die Nutzdaten entfallen.

Sortieren: Formalisierung (cont'd)



Problemstellung

- Gegeben: Ein **Array** von ganzen Zahlen $a[0], \ldots, a[n-1]$ (oder $a[1], \ldots, a[n]$).
- Gesucht: Eine **Permutation** $\pi : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, n-1\}$ so dass $a[\pi(0)] \leq a[\pi(1)] \leq \ldots \leq a[\pi(n-1)]$
- $\pi(i) =$ Welche Position des Ausgangs-Arrays steht an Position i im sortierten Array? **Beispiel** (rechts): $\pi = (3, 6, 8, 4, 0, 7, 1, 9, 2, 5)$

Anmerkungen

- Die obige Problemstellung nennen wir **aufsteigendes** Sortieren. Bei **absteigendem** Sortieren fordern wir

$$a[\pi(0)] \geq a[\pi(1)] \geq \ldots \geq a[\pi(n-1)].$$

- Wir messen die **Laufzeit** von Sortieralgorithmen als die Anzahl der (1) Vergleiche von Array-Werten und/oder (2) Lese-/Schreibzugriffe auf das Array.

Sortieren: Grundannahmen

1. Internes Sortieren
   - Wir nehmen an: Alle Daten befinden sich im **Hauptspeicher**
     $\rightarrow$ Lese- und Schreibzugriffe auf $a[i]$ sind günstig ($O(1)$).
   - **Gegenteil**: **Externes** Sortieren: Die Daten befinden sich auf einem externen, langsamen Speicher[2] *(z.B. Festplatte)*.
2. Günstige Vergleiche
   - Eine Auswertung der Ordnungsrelation ist günstig ($O(1)$).
   - **Gegenbeispiel**: Stringvergleich ($O(length(string))$).
3. Günstige Schreibzugriffe
   - Die Änderung von Werten (`a[i]=c`) sind günstig ($O(1)$).
   - **Gegenbeispiel**: Datenbanken *(Änderungen triggern ggfs. Reorganisation, $O(n)$)*.

[2] siehe auch **Latency Numbers Every Programmer Should Know** [4].

Outline

Insertionsort

Wir behandeln zunächst einige **einfache**, weniger effiziente Sortierverfahren: **Insertionsort**, **Selectionsort**, **Bubblesort**.

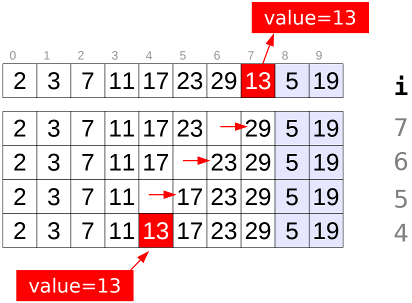
Insertionsort

- Annahme: Die linke Seite des Arrays (a[0],...,a[pos-1]) ist bereits sortiert.
- In jedem Schleifendurchlauf fügen wir a[pos] an der richtigen Stelle in den sortierten Bereich ein.
- Der sortierte Bereich ist jetzt um ein Feld größer (a[1],...,a[pos]).
- Am Ende ist das komplette Array sortiert.

0	1	2	3	4	5	6	7	8	9	**pos**
11	17	23	2	7	29	3	13	5	19	0
11	17	23	2	7	29	3	13	5	19	1
11	17	23	2	7	29	3	13	5	19	2
11	17	23	2	7	29	3	13	5	19	3
2	11	17	23	7	29	3	13	5	19	4
2	7	11	17	23	29	3	13	5	19	5
2	7	11	17	23	29	3	13	5	19	6
2	3	7	11	17	23	29	13	5	19	7
2	3	7	11	13	17	23	29	5	19	8
2	3	5	7	11	13	17	23	29	19	9
2	3	5	7	11	13	17	19	23	29	10

Insertionsort: Pseuco-Code

```
# Gegeben: Array a = a[0],a[1],...,a[n-1]

for pos = 1,.., n-1:

    value = a[pos]

    # Verschiebe alle Werte > value
    # um 1 nach rechts
    for i = pos, pos-1, ..., 0:

        if i>0 and a[i-1]>value:
            a[i] = a[i-1]
        else:
            # Füge value an der
            # richtigen Stelle ein ...
            a[i] = value
            # und beende innere Schleife
            break
```



Ablauf

- In der inneren Schleife (i) werden – beginnend bei pos – alle Elemente größer a[pos] um **eins nach rechts** verschoben.
- a[pos] wird an der **richtigen Stelle** wieder eingefügt.
- Die innere Schleife wird verlassen, wir erhöhen pos um 1.

Insertionsort: Aufwands-Analyse

- Wir berechnen die **Worst-Case**-Laufzeit von Feldelementen! (= Anzahl der **Vergleiche**) f(n).
- Es sei $g_{\text{inner}}(\text{pos})$ die Anzahl der Vergleiche in **einem Durchlauf der inneren Schleife** *(in Abhängigkeit von pos)*.

```
# Gegeben: Array a = a[0],a[1],...,a[n-1]

for pos = 1,.., n-1:

    value = a[pos]

    # Verschiebe alle Werte > value
    # um 1 nach rechts
    for i = pos, pos-1, ..., 0:

        if i>0 and a[i-1]>value:
            a[i] = a[i-1]
        else:
            # Füge value an der
            # richtigen Stelle ein ...
            a[i] = value
            # und beende innere Schleife
            break
```

!

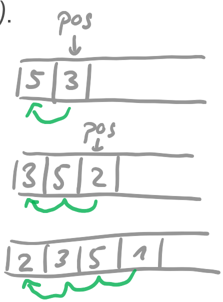
$\text{pos}=1: \; g_{\text{inner}}(1) = 1$

$\text{pos}=2: \; g_{\text{inner}}(2) = 2$

$\text{pos}=3: \; g_{\text{inner}}(3) = 3$

$\vdots$

$\text{pos}=n-1: \; g_{\text{inner}}(n-1) = n-1$



$$\boxed{g_{\text{inner}}(\text{pos}) = \text{pos}}$$

Insertionsort: Aufwands-Analyse

Gesamtaufwand:

$$f(n) = \sum_{pos=1}^{n-1} g_{inner}(pos)$$

|| (letzte Seite)

$$= \sum_{pos=1}^{n-1} pos$$

$$= \frac{(n-1)\cdot n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in \Theta(n^2)$$

```
# Gegeben: Array a = a[0],a[1],...,a[n-1]

for pos = 1,.., n-1:

   value = a[pos]

   # Verschiebe alle Werte > value
   # um 1 nach rechts
   for i = pos, pos-1, ..., 0:

        if i>0 and a[i-1]>value:
             a[i] = a[i-1]
        else:
             # Füge value an der
             # richtigen Stelle ein ...
             a[i] = value
             # und beende innere Schleife
             break
```

Insertionsort: Aufwands-Analyse

Insertionsort: Diskussion

- Insertionsort besitzt **quadratische Komplexität**. ☹
- **Beispiel**: Benchmark einer Java-Implementierung *(Insertionsort auf Zufallszahlen)*.
- Verzehnfacht sich n, verhundertfachen sich (grob) Vergleiche+Änderungen.

Feldlänge n	#Vergleiche	#Änderungen	Laufzeit (Sek.)
10	62 Vergleiche	27 Änderungen	0.00 secs
100	5124 Vergleiche	2516 Änderungen	0.00 secs
1000	505613 Vergleiche	252311 Änderungen	0.02 secs
10000	50393577 Vergleiche	25191792 Änderungen	0.09 secs
20000	200660443 Vergleiche	100320226 Änderungen	0.17 secs
40000	797198244 Vergleiche	398579124 Änderungen	1.15 secs
60000	1806771971 Vergleiche	903355989 Änderungen	1.31 secs
100000	5012503054 Vergleiche	2506201529 Änderungen	3.72 secs

Was macht Insertionsort aufwändig?

- Insertionsort benötigt im Vergleich zu anderen Verfahren **viele Schreibzugriffe**!
- Wird das Element a[pos] ganz vorne eingefügt wird, erfolgen pos Schreibzugriffe!

Outline

Selectionsort

Ansatz

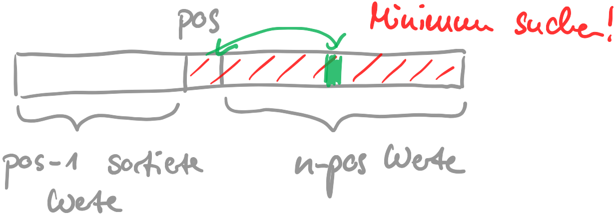
- Tausche das **kleinste** Element an Position 0.
- Tausche das **nächstkleinste** Element an Position 2.
- Tausche das **nächstkleinste** Element an Position 3.
- ...

Pseudo-Code

```
# Gegeben: Array a = a[0],a[1],...,a[n-1]

for pos = 0,.., n-1:

    min := die Position des Minimums von
             a[pos],a[pos+1],...,a[n-1]

    Vertausche a[pos] und a[min]
```

0	1	2	3	4	5	6	7	8	9	pos
11	17	23	2	7	29	3	13	5	19	
11	17	23	2	7	29	3	13	5	19	0
2	17	23	11	7	29	3	13	5	19	1
2	3	23	11	7	29	17	13	5	19	2
2	3	5	11	7	29	17	13	23	19	3
2	3	5	7	11	29	17	13	23	19	4
2	3	5	7	11	29	17	13	23	19	5
2	3	5	7	11	13	17	29	23	19	6
2	3	5	7	11	13	17	29	23	19	7
2	3	5	7	11	13	17	19	23	29	8
2	3	5	7	11	13	17	19	5	29	9
2	3	5	7	11	13	17	19	23	29	10

Selectionsort: Aufwands-Analyse



pos	#Vergleiche	#Vertauschungen
0	n	1
1	$n-1$	1
$\vdots$	$\vdots$	$\vdots$
$n-1$	1	1
Gesamtaufwand	$\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2} \in \Theta(n^2)$	n

Selectionsort: Aufwands-Analyse

Selectionsort benötigt

Outline

Bubblesort: Ansatz

- Es sei `pos=n-1` eine "Endposition".
- Wir durchlaufen das Array von vorne bis `pos`. Immer wenn zwei **benachbarte Elemente** in falscher Reihenfolge sind, **vertauschen** wir sie.
- Hierbei "steigt" das größte Element des Arrays **wie eine Blase** an die letzte Stelle.
- Wir wiederholen den Durchlauf, diesmal bis `pos = n-1`. Danach befindet sich das zweitgrößte Element an vorletzter Stelle.
- Wir wiederholen den Durchlauf mit `pos = n-2`.
- ...

pos=9 (Ende)

0	1	2	3	4	5	6	7	8	9	i =
17	11	23	2	7	29	3	13	5	19	1
11	17	23	2	7	29	3	13	5	19	
11	17	23	2	7	29	3	13	5	19	2
11	17	23	2	7	29	3	13	5	19	3
11	17	2	23	7	29	3	13	5	19	
11	17	2	23	7	29	3	13	5	19	4
11	17	2	7	23	29	3	13	5	19	
11	17	2	7	23	29	3	13	5	19	5
11	17	2	7	23	29	3	13	5	19	6
11	17	2	7	23	3	29	13	5	19	
11	17	2	7	23	3	29	13	5	19	7
11	17	2	7	23	3	13	29	5	19	
11	17	2	7	23	3	13	29	5	19	8
11	17	2	7	23	3	13	5	29	19	
11	17	2	7	23	3	13	5	29	19	9
11	17	2	7	23	3	13	5	19	29	

Bubblesort: Pseudo-Code

```
# Gegeben: Array a = a[0],a[1],...,a[n-1]

Für alle Endpositionen pos = n-1, n-2,... 1:

   Durchlaufe das Array von i = 1 bis pos

   An jeder Stelle i:
      Falls a[i] < a[i-1]:
         Vertausche beide.
```

pos=8 (Ende)

0	1	2	3	4	5	6	7	8	9
11	17	2	7	23	3	13	5	19	29
11	2	17	7	23	3	13	5	19	29
11	2	7	17	23	3	13	5	19	29
11	2	7	17	3	23	13	5	19	29
11	2	7	17	3	13	23	5	19	29
11	2	7	17	3	13	5	23	19	29
11	2	7	17	3	13	5	19	23	29

pos=7 (Ende)

0	1	2	3	4	5	6	7	8	9
11	2	7	17	3	13	5	19	23	29
2	11	7	17	3	13	5	19	23	29
2	7	11	17	3	13	5	19	23	29
2	7	11	3	17	13	5	19	23	29
2	7	11	3	13	17	5	19	23	29
2	7	11	3	13	5	17	19	23	29

...

Bubblesort: Optimierung

- Gegebenenfalls ist das Array bereits ab einer Position $\text{pos}' < \text{pos}$ sortiert.
- **Woran erkennen wir dies?** → Ab Position pos' wurde **kein Tausch** mehr durchgeführt!
- **Was bringt das?** Wir können pos sofort auf pos' setzen und uns einige Iterationen sparen!

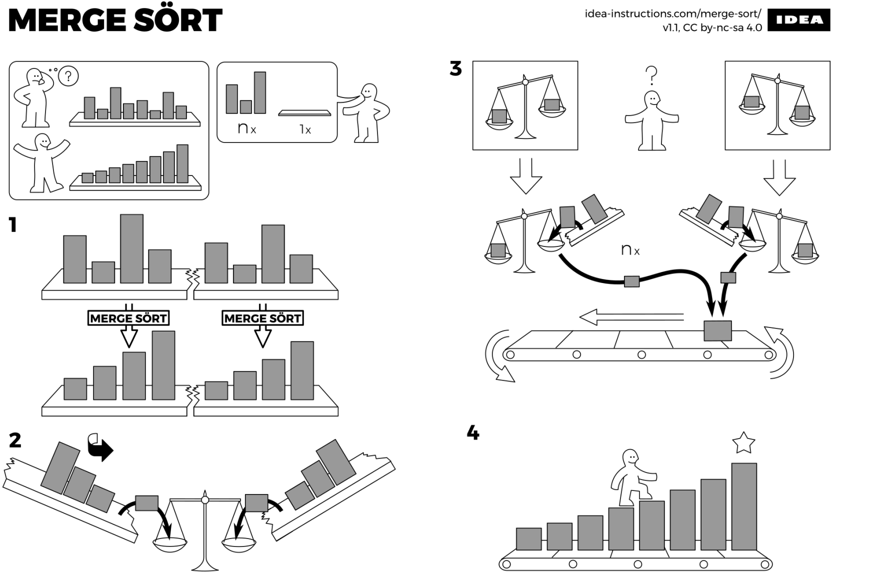
```
# Gegeben: Array a = a[0],a[1],...,a[n-1]

pos = n-1

Wiederhole:

   Durchlaufe das Array von i=1 bis pos

   pos' = 0

   An jeder Stelle i:
      falls a[i] < a[i-1]:
         Vertausche beide.
         pos' = i-1

   pos = pos'

bis pos < 1
```

Bubblesort: Aufwands-Analyse

Bubblesort: Aufwands-Analyse

Outline

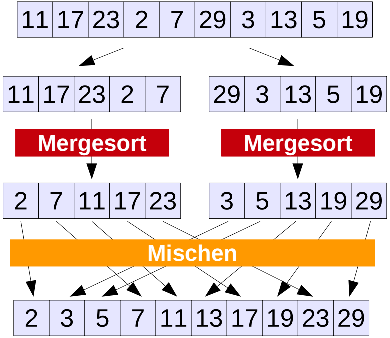
Mergesort



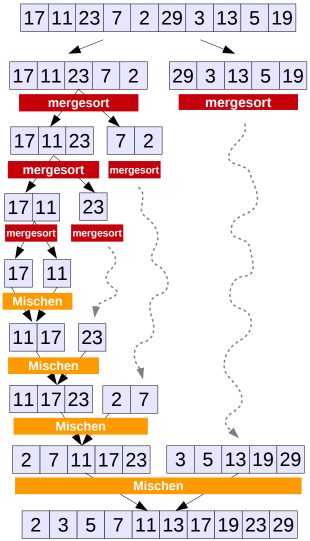
MERGE SÖRT
idea-instructions.com/merge-sort/
v1.1, CC by-nc-sa 4.0
IDEA
n x
1 x
1
MERGE SÖRT
MERGE SÖRT
2
3
n x
4

Mergesort

- Die bisher vorgestellten Suchverfahren besitzen eine **Worst-Case-Komplexität von $O(n^2)$**. ☹
- Wie erreichen wir eine **bessere Komplexität**?
- Idee *(siehe binäre Suche)*: **Divide-and-Conquer**. Zerlege das Problem **rekursiv** in zwei kleinere Teilprobleme.
- **Sortiere die Hälften** des Arrays.
- Die sortierten Hälften **mischen** wir abschließend ineinander.



Mergesort: Pseudo-Code




```
# Subroutine: sortiert Teilbereich
# a[left...right]
function mergesort(a, left, right):

    if left >= right: return

    m = (left + right) / 2

    mergesort(a, left, m)
    mergesort(a, m+1, right)

    Mische die Teilbereiche [left...m]
    und [m+1...right] in [left...right]


# sortiere das ganze Feld
mergesort(a, 0, n-1)
```

- Bestimmung der Mitte `m`.
- Rekursiver Aufruf für beide Hälften
- Mischen der beiden Hälften.
- Gesamt-Sortierung mit `mergesort(a, 0, n-1)`.

Mergesort: Misch-Operation

Wie teuer ist das **Mischen** in Mergesort?

Mergesort: Misch-Operation

Mergesort: Analyse

Mergesort: Analyse

Mergesort: Analyse

Mergesort: Fazit

Zusammenfassung

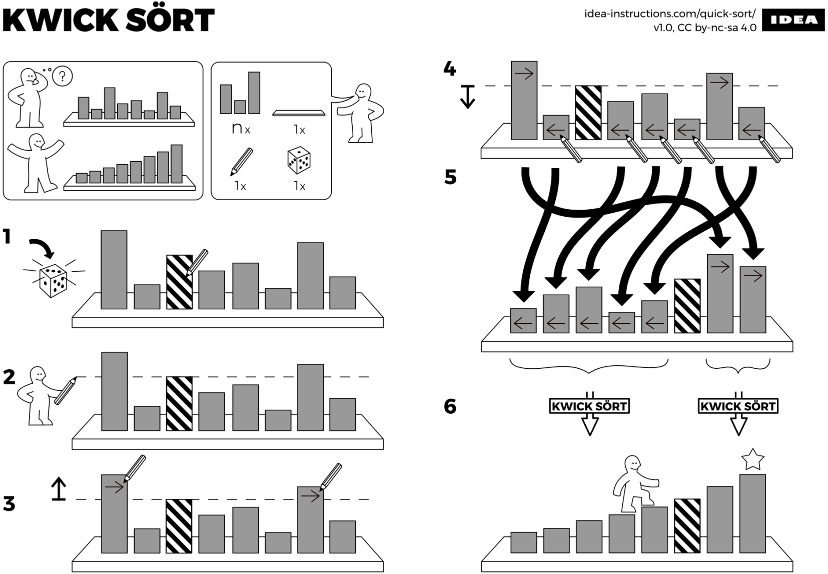
- Die Misch-Operation eines Teilbereichs kostet $O(n)$.
- Rekurrenzgleichung: $T(n) = 2 \cdot T(n/2) + 2n$.
- Komplexität: $O(n \cdot log\ n) \rightarrow$ **Viel, viel besser als $O(n^2)$**
  ($O(n \cdot logn)$ gilt übrigens auch im Best Case + Average Case).

Fazit

- Gutes Allround-Sortierverfahren für alle Fälle.
- Benötigt **Extra Speicher** ($O(n)$) als Puffer
  bei der Misch-Operation.

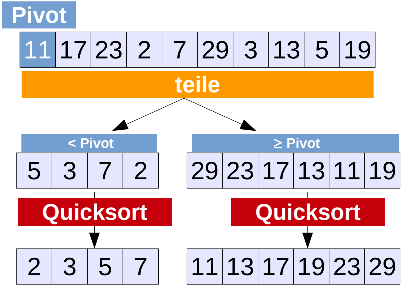
Outline

Quicksort


KWICK SÖRT
idea-instructions.com/quick-sort/
v1.0, CC by-nc-sa 4.0
IDEA
n x
1x
1x
1x
1
2
3
4
5
6
KWICK SÖRT
KWICK SÖRT

Quicksort

Quicksort *(C.A.R. Hoare, 1962)* ist in der Praxis häufig das **schnellste** unter den populären Sortierverfahren.



Grundidee

- Ebenfalls **Divide-and-Conquer** *(siehe Mergesort)*.
- Wähle ein sogenanntes **Pivot-Element**.
- Bewege alle Werte in die "**richtige Hälfte**" *(links: $<$ Pivot. Rechts: $\geq$ Pivot)*.
- Sortiere die beiden Teile rekursiv.
- Es ist kein Merge nötig!

Quicksort: Pseudo-Code (hier: Variante nach Hoare)

- Das **Pivot** ist der Wert **ganz links**.
- **Subroutine teile()**: teilt den Bereich in kleine Werte (links) und große Werte (rechts). Nutzt **zwei Zeiger** `i` und `j`:
  - `i` sucht von links kommend nach Werten ≥ `pivot`
  - `j` sucht von rechts kommend nach Werten < `pivot`
- **Ende sobald beide Zeiger sich treffen.**

```
# sortiert Teilbereich a[left...right]
function quicksort(a, left, right):

    if left >= right: return

    pivot = a[left]

    # Teile den Bereich
    # a[left...right], so dass:
    # - a[left...m] < pivot
    # - a[m+1...right] >= pivot
    m := teile(a, left, right, pivot)

    quicksort(a, left, m)
    quicksort(a, m+1, right)

# sortiere das ganze Feld
quicksort(a, 0, n-1)
```

```
# teilt Teilbereich in kleine und große Werte
function teile(a, left, right, pivot):

  i = left - 1
  j = right + 1

  while true:

    # Suche nächste Positionen:
    # - i sucht große  Werte in linker  Hälfte
    # - j sucht kleine Werte in rechter Hälfte
    do i += 1 while i<=right && a[i] < pivot
    do j -= 1 while j>=left  && a[j] > pivot

    if i>=j:
      return j

    swap a[i] and a[j]
```

Quicksort: `teile()` – Beispiel

Pivot-Element = 11

11	17	23	2	7	29	3	13	5	19

```
# teilt Teilbereich in kleine und große Werte
function teile(a, left, right, pivot):

  i = left – 1
  j = right + 1

  while true:

    # Suche nächste Positionen:
    # - i sucht große  Werte in linker  Hälfte
    # - j sucht kleine Werte in rechter Hälfte
    do i += 1 while i<=right && a[i] < pivot
    do j -= 1 while j>=left  && a[j] > pivot

    if i>=j:
      return j

    swap a[i] and a[j]
```

Quicksort: Beispiel

Quicksort: Beispiel

Quicksort: Komplexität

Wir leiten die Komplexität des Quicksort-Algorithmus her und verwenden hierzu (erneut) **Rekurrenzgleichungen**.

Worst Case

Quicksort: Komplexität

Quicksort: Komplexität

Best Case

Quicksort: Komplexität

Quicksort: Fazit

Komplexität

- Worst Case: $\Theta(n^2)$ *(Pivot immer größes Element)*.
- Best Case: $\Theta(n \cdot log(n))$ *(immer Halbierung des Feldes)*.
- Average Case: $\Theta(n \cdot log(n)) \rightarrow$ *Im Mittel wirklich "quick"*[3].

Schlüssel: Wahl eines "guten" Pivots

- Option: zufällig *(Monte Carlo – Quicksort)*
- Option: Median aus erstem, letztem und mittleren Wert
- ...

[3] R. Sedgewick: Algorithmen in {C, C++, Java}

Quicksort: Fazit (cont'd)

Vorteile in der Praxis

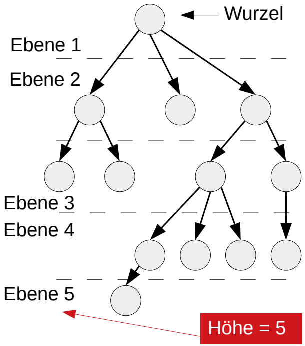
- Quicksort ist in der Praxis häufig das **schnellste** der hier behandelten Verfahren.
  - Die innere Schleife bestehe aus zwei kohärenten Felddurchläufen *(gute Lokalitätseigenschaften, schnell)*
  - Vergleich gegen einen festen Wert, das Pivot *(schnell!)*
- Im Gegensatz zu Mergesort benötigt Quicksort **keinen Extra-Speicher** *(außer dem Call Stack für die rekursiven Aufrufe, $O(n)$ im Worst Case)*.

Outline

Heapsort[4]

Heapsort, unser letzter vergleichsbasierter Sortieralgorithmus

- Worst-Case-Komplexität von $O(n \cdot log(n))$ ☺
- kein zusätzlicher Speicher ☺
- basiert auf einer speziellen Datenstruktur, dem **Heap**.

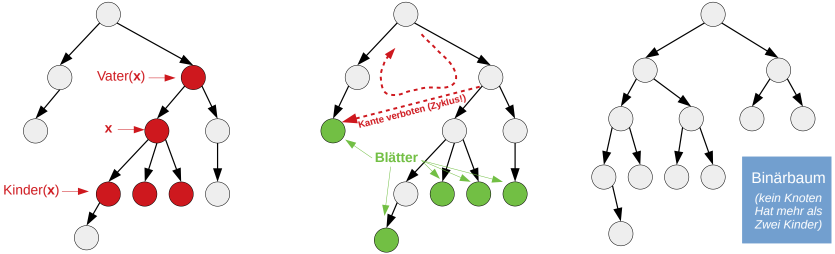


Bäume

- Heaps sind eine spezielle Art von **Bäumen** *(später mehr)*.
- Bäume bestehen aus Knoten ($=$ Datenelementen) und Kanten.
- Die Knoten sind in **Ebenen** angeordnet. Auf Ebene 1 befindet sich die **Wurzel**.
- Die **Höhe** (bzw. **Tiefe**) des Baums entspricht der maximalen Ebene.

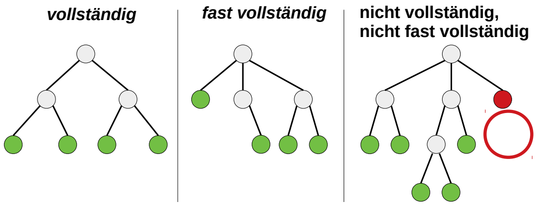
[4] Cormen et al.: Algorithmen – eine Einführung. Oldenbourg-Verlag, 2004.

Exkurs: Bäume



- Alle Knoten *(außer der Wurzel)* besitzen einen **Elternknoten** *(oder Vaterknoten)* und sind ihrerseits **Kinder** dieses Knotens.
- Wir unterscheiden zwischen **Blättern** und **inneren Knoten**.
  Innere Knoten besitzen Kinder, Blätter nicht.
- Bäume enthalten **keine Zyklen**.
- In einem **binären Baum** haben Knoten *maximal zwei Kinder*.

Exkurs: Bäume



Vollständigkeit

Vollständige Bäume besitzen nur auf der letzten Ebene Blätter, **Fast vollständige Bäume** auf den letzten zwei Ebenen.

Die Höhe eines vollständigen Baums ist logarithmisch!

- Was ist die Höhe eines **vollständigen binären Baums** mit **n Elementen**?
- Ein Baum der Höhe h kann $n = 2^h - 1$ Elemente speichern *(Beweis: Induktion)*.
- Gleichung umstellen $\rightarrow h = log_2(n+1)$.
- **Zentrale Eigenschaft von Bäumen: Vollständige Bäume mit sehr sehr vielen Elementen sind ziemlich flach!** *(Bsp. 1,000,000 Elemente $\rightarrow$ Höhe 20)*.

Höhe h	#Elemente n
1	1
2	3
3	7
4	15
5	31
...	...
10	1023
...	...
h	2^h-1

Heaps

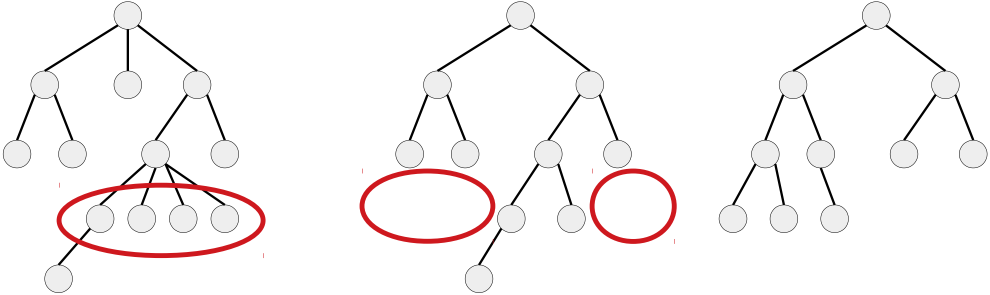
Heaps sind eine besondere Form von Bäumen:

- Sie sind **binär** (jeder Knoten besitzt ≤ 2 Kinder).
- Sie sind **fast vollständig** *(die unterste Ebene wird von links aufgefüllt!)*.
- Sie erfüllen die **Heap-Eigenschaft** *(gleich mehr)*.

kein Heap, denn **nicht binär**

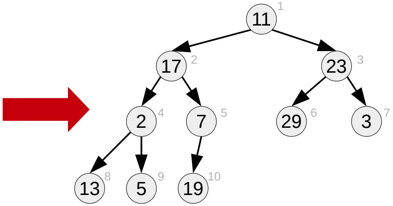
kein Heap, denn **nicht fast vollständig**

Könnte **ein Heap sein**



Heaps (cont'd)

1	2	3	4	5	6	7	8	9	10
11	17	23	2	7	29	3	13	5	19



Wir stellen uns das zu sortierende **Array** $a[1], \ldots, a[n]$ als **Heap** vor!

- Element Nr. 1 wird zur Wurzel
- Elemente Nr. 2-3 bilden die 1. Ebene
- Elemente Nr. 4-7 bilden die 2. Ebene
- ...

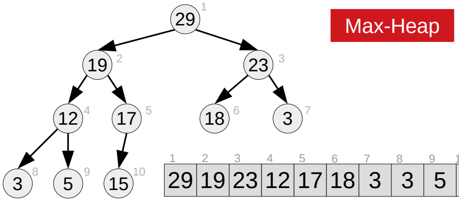
Mit folgenden einfachen Funktionen greifen wir auf den **Elternknoten** / die **Kinder** eines Elementes i zu:

$$parent(i) = \lfloor i/2 \rfloor$$
$$left\_child(i) = 2 \cdot i$$
$$right\_child(i) = 2 \cdot i + 1$$

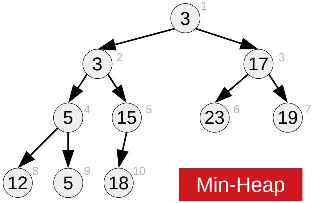
Heaps (cont'd)

Heaps sind also binäre, fast vollständige Bäume. Zusätzlich erfüllen sie noch die folgende **Heap-Eigenschaft** (sie sind **teilsortiert**):

> **Definition (Heap-Eigenschaft)**
> *Wir interpretieren ein Array $a[1], \ldots, a[n]$ als binären Baum. Wir bezeichnen das Array als* **Min-Heap** *(bzw.* **Max-Heap***), wenn für alle i mit $1 < i \le n$ gilt:*
>
> $a[parent(i)] \ge a[i]$ **(Max-Heap)**
>
> $a[parent(i)] \le a[i]$ **(Min-Heap)**



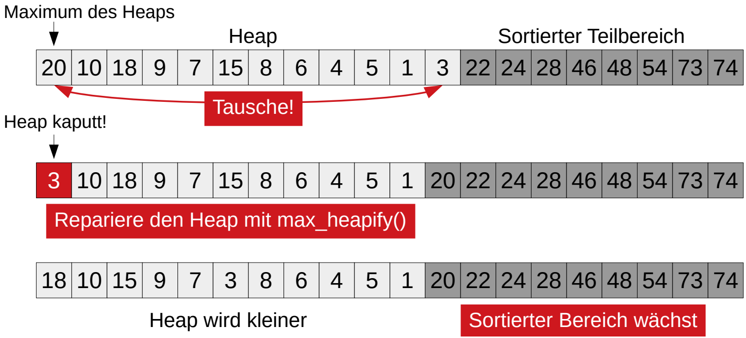
1	2	3	4	5	6	7	8	9	10
29	19	23	12	17	18	3	3	5	15



Heapsort: Grundidee

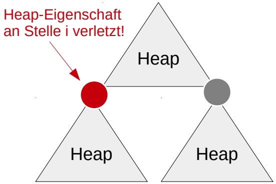
- Verwandle das zu sortierende Array in einen **Max-Heap**.
- Entnehme in jeder Iteration die **Spitze** des Heaps *(= das Maximum!)*, tausche es zum **Ende** des Arrays.
- Stelle für den Rest des Heaps die **Heap-Eigenschaft** wieder her.

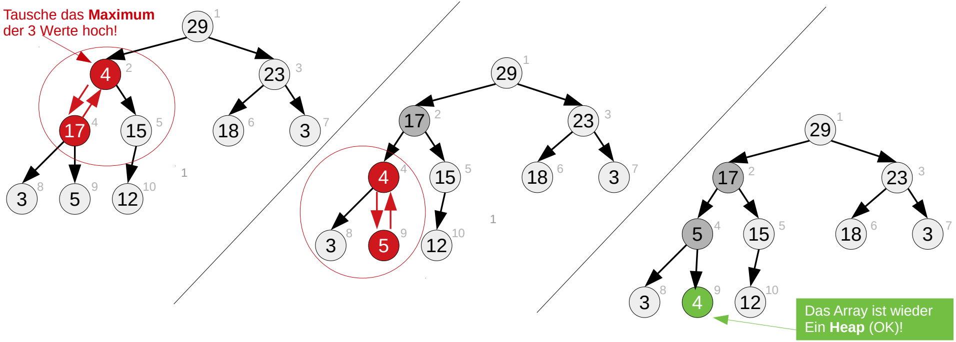
```
method heapsort(a):

    # turn a into a heap
    build_max_heap(a)

    for i = n, n-1, ..., 2:

        swap(a[1], a[i])

        # decrease the heap by 1
        heapsize -= 1

        # repair the heap
        max_heapify(a, 1)
```



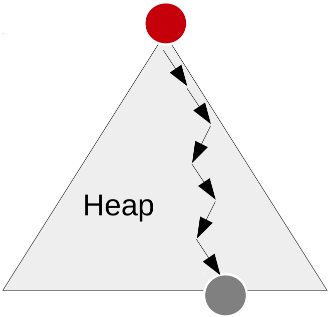
Heapsort: `max_heapify(a)`

- **Ausgangssituation**: Das Array a ist <u>fast</u> ein Heap: Nur an Stelle `i` ist die Heap-Eigenschaft verletzt.
- **Ansatz**: Lasse das “falsche” Element rekursiv an die korrekte Position sinken.





`max_heapify()`: Komplexität



Wie teuer ist ein Aufruf von `max_heapify()` im Worst Case?

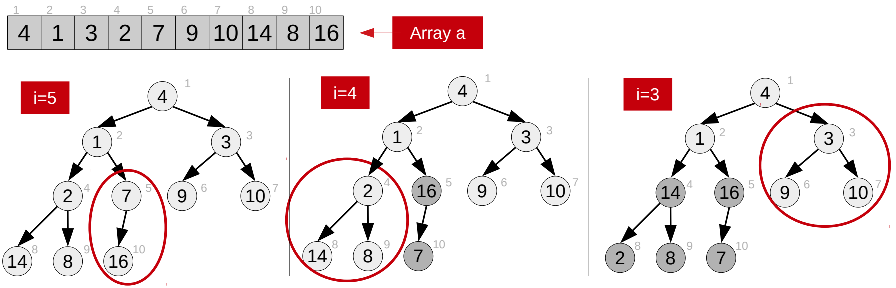
- **Worst Case**: Ein Element sinkt bis nach ganz unten.
- **Je Ebene**: O(1) *(3 Werte vergleichen, 2 Werte tauschen)*.
- **#Ebenen**: Höhe des Baums → $O(log(n))$.
- **Gesamtaufwand:** $O(log(n))$.

build_max_heap()

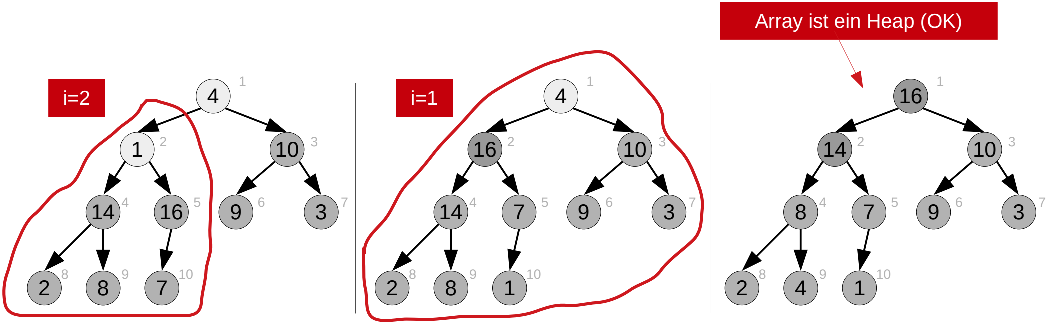
```
method build_max_heap(a):

    for i in n/2, ..., 1:
        max_heapify(a, i)
```

Zu **Beginn** verwandeln wir das zu sortierende Array mit `build_max_heap()` in einen **Heap**.

- Wir durchlaufen das Array von hinten nach vorne.
- An jeder Position i stellen wir für den Bereich $a[i], \ldots, a[n]$ die **Heap-Eigenschaft** sicher, indem wir `max_heapify(a, i)` aufrufen.



build_max_heap() (cont'd)



build_max_heap()

```
method build_max_heap(a):

    for i in n/2, ..., 1:
        max_heapify(a, i)
```

Was ist die Komplexität
von build_max_heap()?

- Ein Aufruf von max_heapify() kostet $O(log(n))$.
- Wir rufen $n/2$ (also $O(n)$) mal max_heapify() auf.
- **Gesamtaufwand:** $O(n \cdot log(n))$.
- Anmerkung: Man kann sogar zeigen dass der Aufwand $O(n)$ – d.h. noch günstiger – ist[5].

[5] Cormen: Algorithmen -– eine Einführung. Oldenbourg-Verlag, 2004.

Heapsort: Diskussion

Aufwandsanalyse

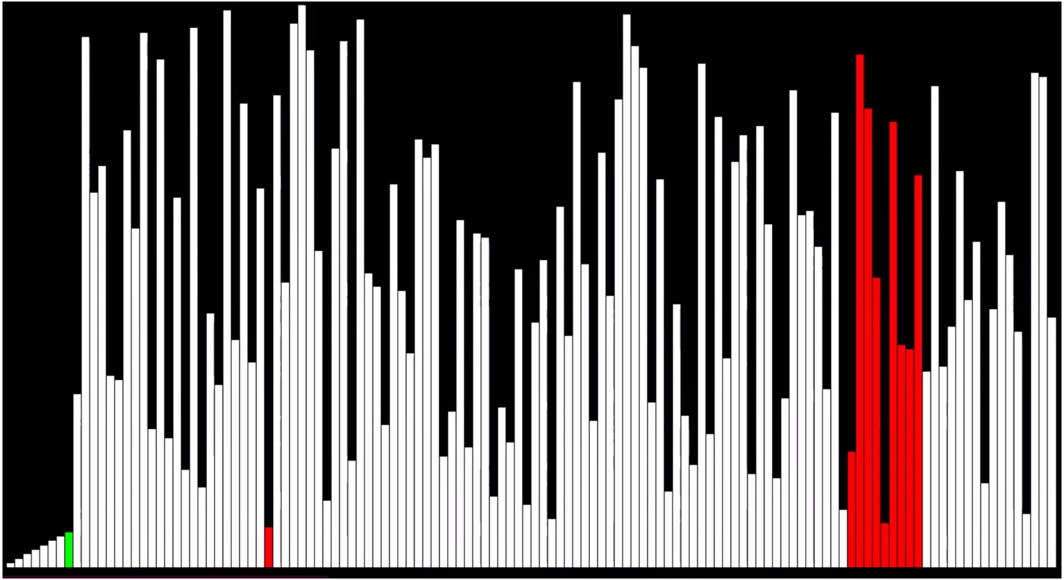
- Aufwand für `build_max_heap()`: $O(n)$.
- Aufwand pro Schleifendurchlauf: $O(1) + O(1) + O(log(n)) = O(log(n))$.
- Aufwand der Schleife (n Durchläufe): $O(n \cdot log(n))$.
- **Gesamtaufwand:** $O(n) + O(n \cdot log(n)) = O(n \cdot log(n))$.

```
method heapsort(a):

O(n)        # turn a into a heap
            build_max_heap(a)

n x         for i = n, ..., 2:

O(1)            swap(a[1], a[i])

O(1)            # decrease the heap
                heapsize -= 1

O(log n)        # repair the heap
                max_heapify(a, 1)
```

Vergleich mit anderen effizienten Sortierverfahren

- Worst-Case: Besser als Quicksort ($O(n^2)$).
- Speicher: Besser als Mergesort
  (Heapsort ist "in-place")
- Quicksort ist in der Praxis oft schneller, weil cache-efizienter.

Sortieren: Video Bild: [1]



Outline

Digitales Sortieren

- Die bisherige Sortierverfahren vergleichen den **kompletten Schlüssel**.
- Im folgenden wollen wir die Strukturinformation des Schlüssels *(d.h. die einzelnen Ziffern)* nutzen *("digitales Sortieren")*.

Digitales Sortieren: Grundannahmen

- Schlüssel sind Zeichenketten über einem Alphabet aus m Elementen *(den einzelnen Ziffern)*.
- Wir nennen m auch die **Wurzel** *(Latein: radix)*.
- Wir nehmen an, dass die Länge der Schlüssel *(und somit ihr Wertebereich)* beschränkt sind.

Radix m	Bezeichnung	Beispiel
10	Dezimalzahlen	72945
16	Hexadezimalzahlen	48fc
2	Binärzahlen	0011101
256	Strings *(extended ASCII)*	hallo.

Radix Exchange Sort

Voraussetzungen

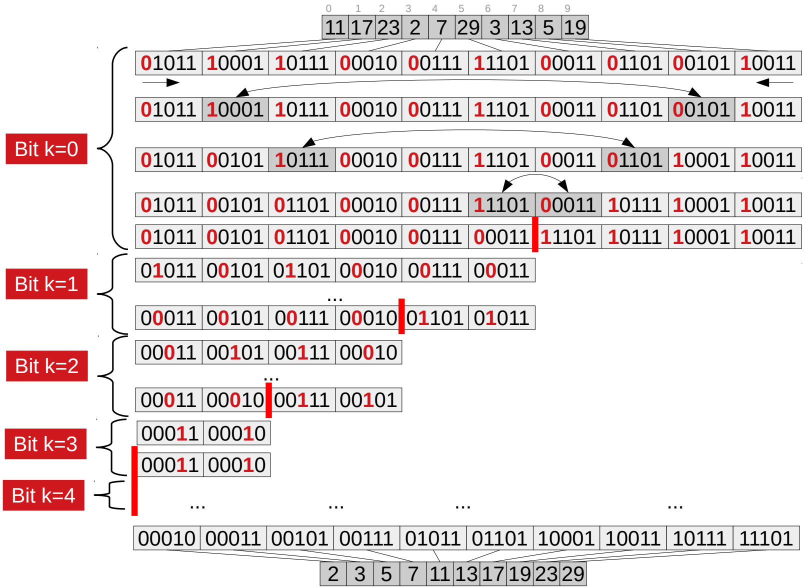
- m-adische Zahlen (hier: **Binärzahlen**) fester Länge K.
- **Beispiel**: 32-Bit-Binärzahlen → 2^{32} verschiedene Schlüssel.

```
function radixsort(a, left, right, k):

    for i = left, ..., right:
        b[i] = k-tes Bit von Element a[i]

    # Teile die Elemente gemäß des k-ten Bits:
    # Nullen nach links und Einsen nach rechts.
    # -> Schema von Hoare (siehe QuickSort)
    m = teile (a, b, left, right)

    if k >= L:
        # letztes Bit erreicht
        return
    else:
        # sortiere nach dem nächsten Bit
        radixsort(a, left, m,     k+1)
        radixsort(a, m+1,  right, k+1)

# ganzen Bereich sortieren
radixsort(a, 0, n-1, 0)
```

Ansatz

- Wir durchlaufen die **einzelnen Ziffern / Bits** $k = 0, \dots, K-1$ der Schlüssel.
- Für jede Ziffer k **partitionieren** wir rekursiv die Schlüssel nach der entsprechenden Ziffer in 0 (links) und 1 (rechts).

Radix Exchange Sort



Radix Exchange Sort: Diskussion

Aufwandsbetrachtung

- Es sei **n die Anzahl** der zu sortierenden Schlüssel und **K die Länge** der Schlüssel.
- Jedes Element des Arrays wird genau K-mal besucht
  (das macht insgesamt $K \times n$ Besuche!).
- Jeder Besuch eines Elements kostet $O(1)$.

→ **Gesamtkomplexität:** $O(n \cdot K)$.

Anmerkungen

- **Achtung**: K ist keine Konstante, d.h. Der Aufwand ist abhängig vom darstellbarem Zahlenbereich!
- Sortieren wir z.B. n unterschiedliche Schlüssel, muss gelten $K \geq log_m(n)$ *(also mindestens $O(n \cdot log(n))$)*.
- Radix Exchange Sort ist ungünstig bei sehr langen Schlüsseln / großen Zahlenbereichen.
  Beispiel: *Sortiere* 10 64-*Bit-Zahlen.*

Outline

Benchmark Sortierverfahren

Einfache vs. schnelle Sortierverfahren in der Praxis: Laufzeit (in Sekunden) für verschiedene Eingabegrößen.

Verfahren n=	1,000	10,000	100,000	800,000	1,000,000
Selektion	<0,01	0,07	12,03	767,53	1200,24
Bubble	<0,01	0,10	16,20	1040,39	1625,77
Insertion	<0,01	0,03	3,48	223,33	348,30
Quick	<0,01	0,01	<0,01	0,08	0,10
Merge	<0,01	0,02	0,01	0,14	0,18
RadixExchange	<0,01	0,02	0,01	0,11	0,14

* java 1.6.0.26, Java HotSpot 64-Bit Server VM, 20.1-b02, mixed mode, i7 2600, 3.4GHz, 8GB RAM, Angaben in Sekunden

Stabilität von Sortierverfahren

Definition (Stabilität)

Wir bezeichnen ein Sortierverfahren als **stabil** *wenn das Verfahren die Reihenfolge* **identischer Schlüssel** *nicht verändert.*

Note	MatNr
1	1001
2	1002
1	1003
1	1004
1	1005
2	1006

OO-Calc, früher nicht stabil

Note	MatNr
1	1003
1	1004
1	1001
1	1005
2	1006
2	1002

Excel, stabil

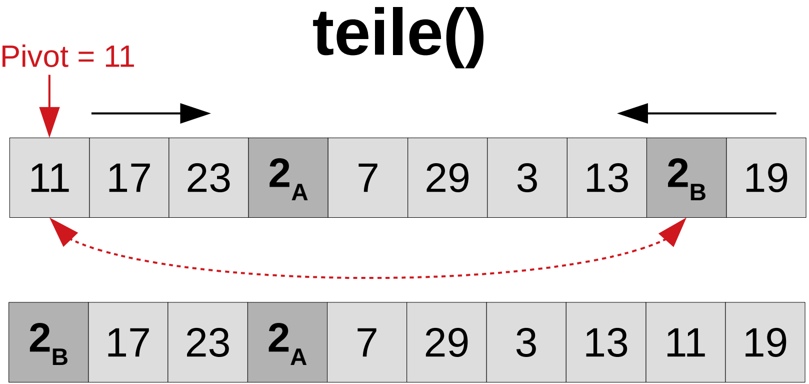
Note	MatNr
1	1001
1	1003
1	1004
1	1005
2	1002
2	1006

Anmerkungen

- **Relevanz**: **Objekte** nicht unerwartet tauschen.
- **Beispiel (links)**: Sortiert man nach der Note, sollte sich bei gleichen Noten die Reihenfolge der Matrikelnummern nicht ändern.

Beispiel Quicksort

- QuickSort ist **nicht stabil**!
- `teile()`: Keine Kontrolle wie Werte links und rechts des Pivots landen.



Sortierverfahren in Java

In der Bibliothek `java.util` bietet die Klasse `Arrays` Sortierverfahren als statische Methoden, z.B. `sort(int[] a)`.

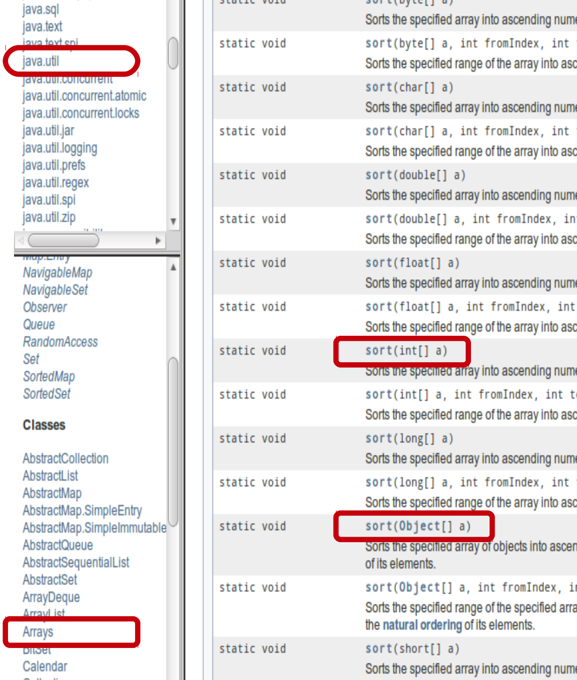
Implementierung?

- Für **short[], int[], long[]**: "Dual-Pivot" QuickSort.
- Für **Objekte**: MergeSort *(stabil)*.

Hybride Verfahren

In der Praxis werden häufig Kombinationen mehrerer Verfahren verwendet.

- Merge-Insertion-Sort [3]
- Introsort [2] *(Quick-/Merge)*
- ...



Outline

Komplexität Algorithmus ≠ Problem

Wiederholung

- Komplexität eines Problems :=
  Komplexität des **besten** Algorithmus.
- **Beispiel**: Lineare Suche $O(n)$, Binäre Suche $O(log\ n)$.

Worst-Case-Komplexität des Problems “Sortieren?

- Wir kennen Algorithmen mit $\Theta(n \cdot log\ n)$ *(z.B. Mergesort)*.
- **Frage: Geht es noch besser?**

Fragestellung: Untere Schranke des Sortierens

- **Untere Schranke** = Worst-Case-Komplexität, die von **keinem** Verfahren **unterschritten** werden kann.
- Eine offensichtliche untere Schranke des Sortierens
  ist $O(n)$ *(wir müssen alle Werte des Arrays besuchen)*.
- **Wir zeigen aber**: Die untere Schranke ist $\boldsymbol{\Theta(n \cdot log(n))}$.

Hier: Vergleichsbasierte Verfahren

Wir betrachten hier nur **vergleichsbasierte Verfahren**,
die auf Schlüsselvergleichen ($a[i] < a[j]$?) beruhen.

Vergleichsbasiert	**Nicht Vergleichsbasiert**
Selectionsort, Bubblesort, Insertionsort, Mergesort, Heapsort, Quicksort	RadixExchangeSort, *Bucketsort*, *Countingsort*

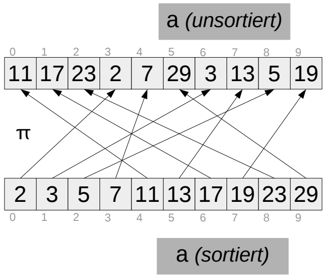
Untere Schranke $\Theta(n \cdot \log(n))$: Beweis

Ansatz

- Wir können nicht <u>alle</u> Sortierverfahren analysieren. ☹.
- Stattdessen führen wir einen Beweis anhand des **Entscheidungsbaums von Sortierverfahren**.

Beweisidee

- Wir erinnern uns: Ziel von Sortieren ist es, eine passende **Permutation π** der Eingabedaten zu finden.
- Es gibt sehr viele Permutationen.
- Ein Sortierverfahren muss also aus **sehr vielen** möglichen Permutationen die **passende** finden.
- Hieraus ergibt sich ein gewisser **Mindestaufwand**.



Wieviele Permutationen gibt es?

Definition (Permutationen)

Gegeben n Zahlen, lautet die Anzahl möglicher Permutationen

$$n! \quad := \quad 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

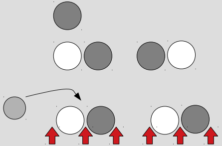
Beweis per vollständiger Induktion (Illustration):

Induktionsanfang

n=1 1 = 1! Permutationen

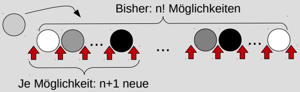
n=2 2 = 2! Permutationen

n=3 3 x 2 = 6 = 3! Permutationen



Induktionsschritt

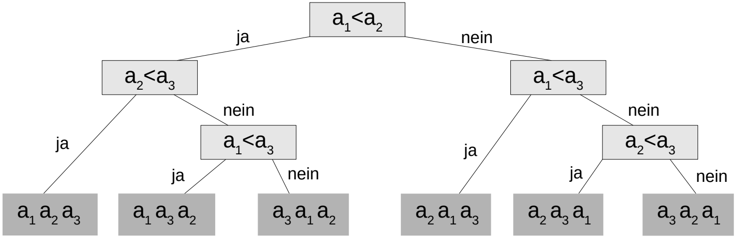
n → n+1 (n+1) x n! = (n+1)! Permutationen



Entscheidungsbäume

- Um die richtige Permutation zu finden, führt das Sortierverfahren **Vergleiche** durch (z.B. $a_3 < a_5$?).
- Wir zählen zur Berechnung der **Laufzeit** diese **Vergleiche**.
- Es entsteht ein **Entscheidungsbaum**.

Beispiel: Sortiere 3 Zahlen



Wichtige Beobachtung: Die Anzahl der **Blätter** entspricht der Anzahl möglicher **Permutationen** (3!=6).

Beweis untere Schranke

- Unterschiedliche Sortierverfahren führen die Vergleiche in unterschiedlicher Reihenfolge durch *(und erzeugen so unterschiedliche Entscheidungsbäume)*.
- Können wir den **Worst-Case-Aufwand** am **Entscheidungsbaum** ablesen?
  $\rightarrow$ Ja: Der Worst-Case-Aufwand entspricht der **Tiefe des Baums**!
- **Wie tief muss der Entscheidungsbaum mindestens sein?**

Beweis untere Schranke

Beweis untere Schranke

Beweis untere Schranke

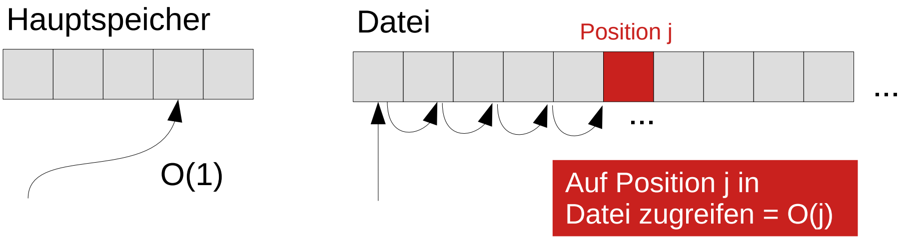
Outline

Motivation: Sehr große Daten sortieren

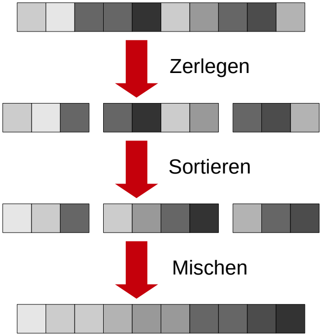
- Bisher: internes Sortieren → das ganze Array befindet sich im **Hauptspeicher**.
- **Bequemer Zugriff** auf alle Elemente in $O(1)$.

Externes Sortieren

- Die Daten seien nun größer als der Hauptspeicher.
- Sie befinden sich z.B. in Dateien auf der Festplatte.
- Hier erfolgt der Zugriff **sequentiell** *(über I/O-Streams)*.



Externes Sortieren



Ansatz

1. Verteile die Daten auf Blöcke
   (jeder so groß wie der Hauptspeicher).
2. Sortiere die einzelnen Blöcke intern.
3. Verteile die Blöcke auf mehrere Dateien.
4. Mische die Dateien *(vgl. Mergesort)*.

Mischen von N Dateien

Ähnlich Mergesort:

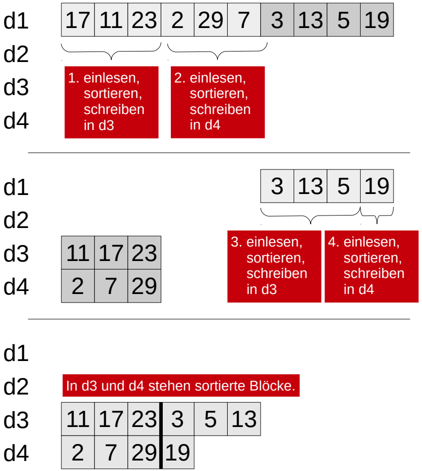
- Lese vorderstes Element jeder Eingabedatei $\rightarrow$ N Werte.
- Wähle den kleinsten Wert, schreibe ihn in die Ausgabedatei.
- Gehe zum nächsten Wert.

Externes Sortieren: Beispiel

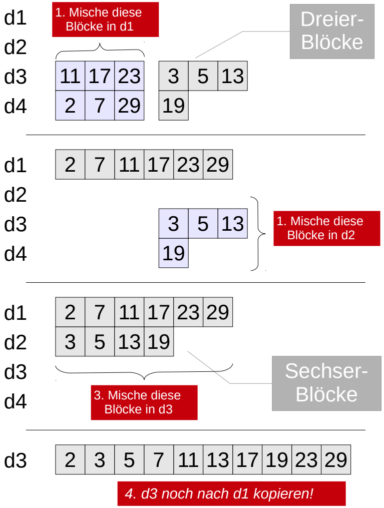
- Wir arbeiten mit 4 Dateien d_1, d_2, d_3, d_4.
- Hauptspeicher: Größe 3.
- Eingabedaten: in Datei d_1.

Schritt 1: In Blöcke zerlegen, Blöcke sortieren

- Lese Daten blockweise und sortiere Blöcke
- Schreibe (sortierte) Blöcke in d3 und d4.



Externes Sortieren: Beispiel



Schritt 2: Mischen

- Mische je **Dreier-Blöcke** aus d_3 und d_4.
- Wir erhalten sortierte Blöcke der **Größe 6**.
- Schreibe diese abwechselnd in d_1 und d_2.
- Wenn wir dieses Mischen **wiederholen**, erhalten wir sortierte 12er-Blöcke, 24er-Blöcke, ...
- Wir mischen bis die kompletten Daten sortiert sind.

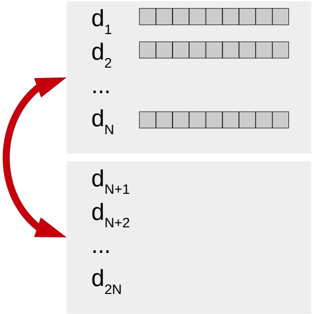
N-Wege-Mischen

Im Beispiel wurde ein sogenanntes **Zwei-Wege-Mischen** durchgeführt:

- Jeder Mischvorgang liest aus **zwei** Dateien *(z.B.* d_1, d_2*)*.
- Jeder Mischvorgang schreibt in **zwei** Dateien *(z.B.* d_3, d_4*)*.

Verallgemeinerung: *N*-Wege-Mischen

- Mische aus *N* Dateien in *N* Dateien *(N Quellen, N Senken)*.
- Je größer *N*, desto **schneller wachsen** die Blöcke und desto **weniger Mischvorgänge** werden benötigt.
- *N* ist in der Praxis **beschränkt**:
  Es gibt eine Obergrenze parallel lesbarer Dateien *(z.B. Linux: "ulimit -n")*.



N-Wege-Mischen: Analyse

Parameter	Definition	Parameter	Definition
n	Anzahl zu sortierender Werte	2N	Anzahl Dateien
H	Kapazität des Hauptspeichers	B := n / H	Anzahl Blöcke

Wir berechnen die Laufzeit

- Vor dem ersten Mischen beträgt die Blockgröße H, nach dem ersten Mischen $N \cdot H$, nach dem zweiten Mischen $N^2 \cdot H$, nach dem zweiten Mischen $N^3 \cdot H$, etc.
- Wieviele Mischvorgänge m benötigen wir bis die Blockgröße den Wert n erreicht?

$$n \stackrel{!}{=} \frac{n}{H}, \text{ also } m = log_N\left(\frac{n}{H}\right)$$

- **Beispiel**: 1 Mrd. Objekte der Größe 1KB (= 1TB), 1 GB Hauptspeicher
  - → Objekte im Hauptspeicher: H = 1GB / 1KB = 1 Mio.
  - → Anzahl Mischvorgänge (N=10) = $log_{10}(n/H) = log_{10}(1000) = 3$.

References I

[1] 15 Sorting Algorithms in 6 Minutes.
https://www.youtube.com/watch?v=kPRA0W1kECg (retrieved: May 2019).

[2] Introsort (Wikipedia).
https://en.wikipedia.org/wiki/Introsort (retrieved: May 2019).

[3] Merge-Insertionsort (Wikipedia).
https://en.wikipedia.org/wiki/Merge-insertion_sort (retrieved: May 2019).

[4] Colin Scott.
Latency Numbers Every Programmer Should Know.
https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html (retrieved: Mar 2018).