Technische Hochschule Rosenheim
Technical University of Applied Sciences

Algorithmen und Datenstrukturen
Kapitel 4: Sortieren

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Über Aufräumen und Sortieren

Quelle[2]

*„Computer manufacturers in the 1960s estimated that more **than 25 percent of the running time of computers were spent sorting**. [...] In fact, there were many installations in which tasks of sorting were responsible for more than half of the computing time.“*

(Donald E. Knuth)

Inhalt

Sortieren vs. Suchen

Suchproblem
Finde ein bestimmtes Element einer Menge

(Kapitel 5, 6, 7)

Sortierproblem
Sortiere eine Menge der „Größe“ nach

(dieses Kapitel 4)

- In einer sortierten Menge ist das Suchen leichter!

Sortierproblem

- **Gegeben:**
  - Folge $< a_1, a_2, \ldots, a_n >$ von *n* Elementen.
  - Jedes Element a_i hat Schlüssel (Key) k_i

- **Gesucht:**
  - Permutation $< a'_1, a'_2, \ldots, a'_n >$ der Eingabe
  - Es muss dann gelten: $k'_1 \leq k'_1 \leq \cdots k'_n$

- **Eigenschaften von Schlüsseln**
  - Sind vergleichbar: Ordnungsrelation $\leq$
  - Meist ganzzahlig
  - Typ spielt prinzipiell keine Rolle, solange zwei Schlüssel durch $\leq$ eindeutig vergleichbar sind.
  - *Frage*: Wie kann man in Java zwei Objekte eines Typs vergleichen?

    → `compareTo, compare, equals`

Bewertung von Sortierverfahren

- **Mögliche Anforderungen**
  - Wenig Vergleiche
  - Wenig Zuweisungen, Anweisungen
  - Wenig Speicherplatz
- **Internes** vs. **externes** Sortieren
  - *Intern*: Die zu sortierenden Daten finden im Hauptspeicher Platz (→ beliebiger Zugriff)
  - *Extern*: Daten auf externem Speicher (Bänder, Festplatten) (→ *nur* sequentieller Zugriff)
- **In-place** vs. **out-of-place**
  - *In-place* : Sortieralgorithmus arbeitet direkt auf Eingabe, kein zusätzlicher Speicherplatz.
- **Vergleichsbasierte** vs. **spezielle** Sortierverfahren
  - *Vergleichsbasiert*: Es werden nur Schlüsselvergleiche ($\leq$) zum Sortieren verwendet.
  - *Speziell*: Ausnutzen spezieller Eigenschaften, z.B. ganze Zahlen als Schlüssel (Radixsort, Countingsort)
- Spezielle Sortierverfahren für **teilweise vorsortierte Daten**

Laufzeitanalyse: Meist genügt es, die Anzahl der Vergleiche zu zählen!

Wiederholung: Insertionsort

```
INSERTION-SORT(A)                                    // A: Input Array
1    for j = 1 to A.length-1
2       key = A[j]
3       // insert A[j] into already sorted sequence
4       i = j - 1
5       while i ≥ 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8       A[i+1] = key
```

Quellcode: InsertionSort.java
(siehe Kapitel 01: Grundlagen)

- ***Animation***
  - https://algorithm-visualizer.org/brute-force/insertion-sort
- ***Invariante***
  - Nach dem j.-ten Durchlauf ist das linke Teilarray mit $j + 1$ Elementen bereits sortiert.
- ***Anzahl der Kopiervorgänge***
  - $O(n^2)$: Gleiche Größenordnung wie Anzahl Schlüsselvergleiche.

Wiederholung: Bubblesort

```
BUBBLE-SORT(A)                                    // A: Input Array
1 for i = 1 to A.length - 1
2    for j = A.length - 1 downto i
3       if A[j] < A[j-1]
4          exchange A[j] with A[j-1]
```

Quellcode: siehe Übung 01

- ***Animation***
  - https://algorithm-visualizer.org/brute-force/bubble-sort
  - Achtung: Dort ist nach dem 1. Durchlaufen der äußeren for-Schleife das Maximum an der korrekten Stelle.
- ***Invariante***
  - Nach dem *i*.-ten Schritt ist das *i*.-kleinste Element an der richtigen Stelle.
- ***Anzahl der Vergleiche***
  - In dieser „dummen“ Fassung immer: $\Theta(n^2)$.
  - Wann könnte man ggfs. vorzeitig abbrechen?
- ***Anzahl der Kopiervorgänge***
  - Abhängig von Art der Eingabe.

Publikums-Joker: Bubble vs. Insertionsort

Gegeben sei das folgende Array A = <1,5,1,1>. Die Fragen beziehen sich auf die Implementierungsvarianten von Folie 6 und Folie 7. Welche Aussage ist **falsch*?***

A. Insertionsort führt genauso viele Elementvergleiche durch wie Bubblesort.

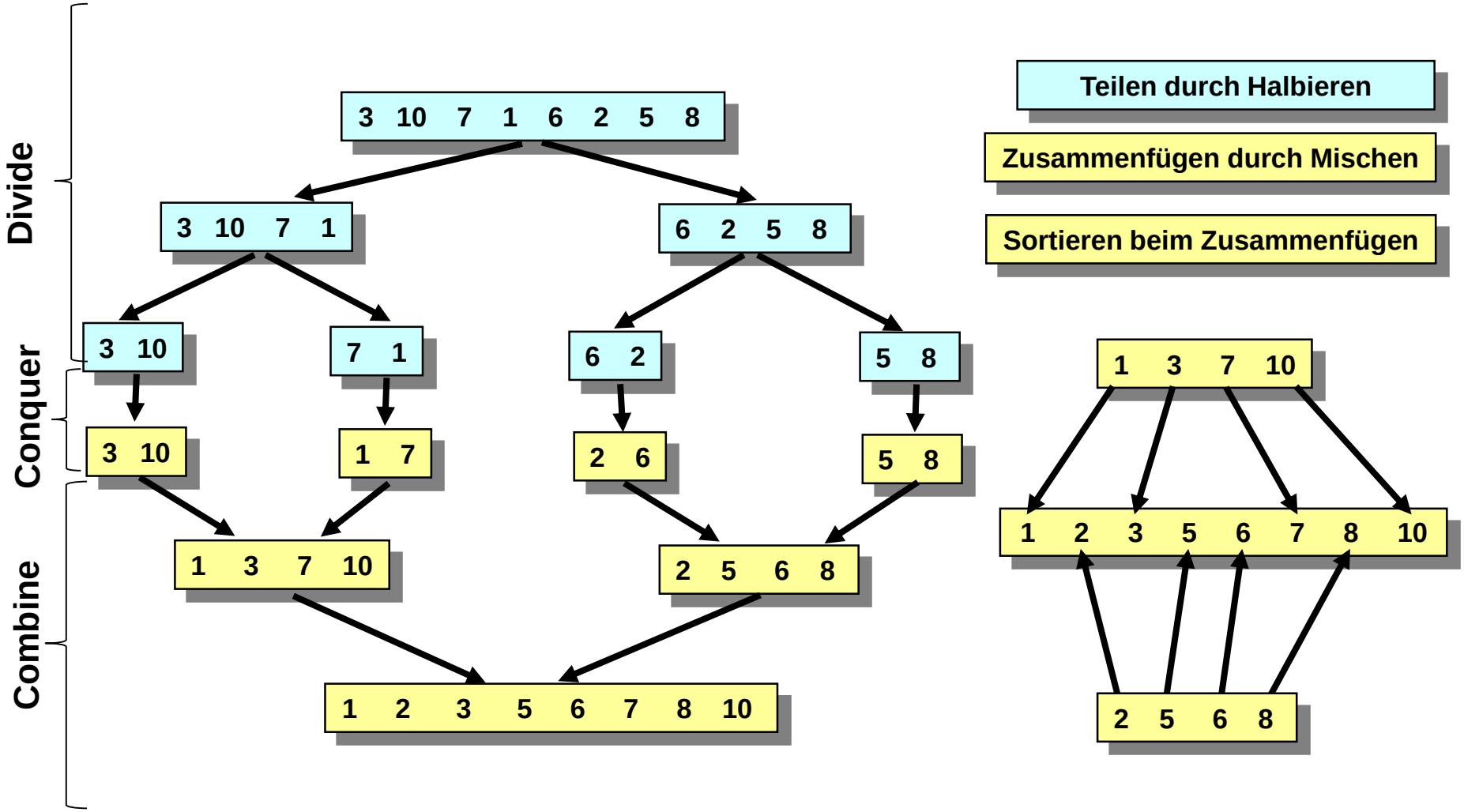
B. Insertionsort führt genauso viele Elementvertauschungen wie Bubblesort durch.

C. Insertionsort und Bubblesort benötigen beide O(1) zusätzlichen Speicherplatz.

Inhalt

- Einführung
- **Mergesort**
- Quicksort
- Heapsort
- Sortieren in linearer Zeit
- Zusammenfassung

Sortieren mit Mergesort

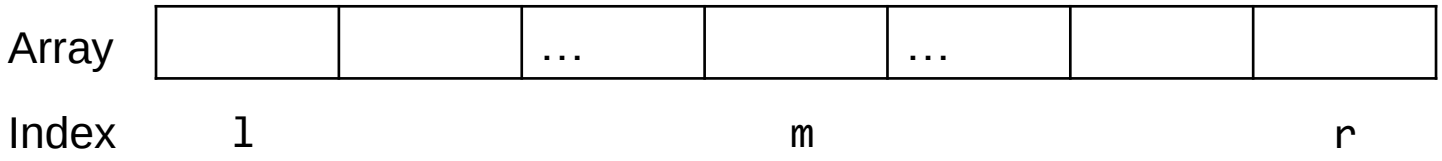


Animation: https://algorithm-visualizer.org/divide-and-conquer/merge-sort

Mergesort - Verfahren

- **Grundidee**
  - Aufteilen der Menge in 2 gleich große Teilmengen.
  - Sortiere beiden Teile, ggfs. rekursiv.
  - Dann mische beide Teilmengen zusammen ("***Merge***“) im ***Reißverschlussverfahren.***

- **Divide-and-Conquer:** Um Array `A[l..r]` zu sortieren
  - ***Divide:*** Teile auf in `A[l..m]` und `A[m+1..r]` mit m als Mitte.
  - ***Conquer:*** Sortiere rekursiv `A[l..m]` *und* `A[m+1..r]`
  - ***Combine****:* ***"***Merge" die beiden nun sortierten Subarrays `A[l..m]` und `A[m+1..r]` um ein sortiertes Array `A[l..r]` zu erhalten.



Mergesort

Wende Algorithmus auf das Teilarray `A[l..r]` an.

```
MERGE-SORT(A, l, r)
1  if l < r
2     m = ⌊(l+r)/2⌋
3     MERGE-SORT(A, l, m)  // div.+conquer links
4     MERGE-SORT(A, m+1, r)// div.+conquer rechts
5     MERGE(A, l, m, r)       // combine
```

"Abrunden": Berechne Index für Mitte

Rekursion

Quellcode: MergeSortRecursive.java

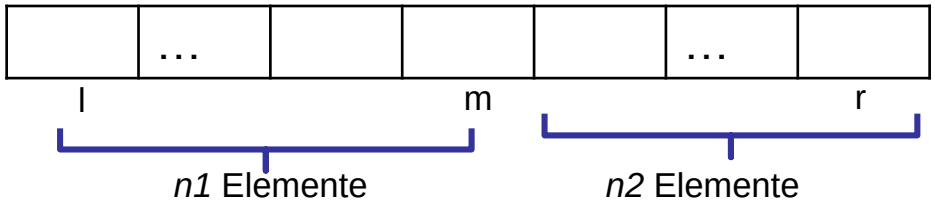
Wie implementiert man MERGE effizient?

- Übung: A=[5 2 4 7 1 3 2 6]
  - Überzeugen, dass die vorgeschlagene Aufteilung funktioniert.
- Kernfunktion
  - Merge: Zusammenführen im "Reißverschlussverfahren"

Merge: Reißverschlussverfahren

- **Idee**
  - "Linkes" und "rechtes" Array sind ***bereits sortiert***.
  - Vergleiche jeweils das linke Element des "linken" und des "rechten" Arrays und sortiere den kleineren der beiden in das Ergebnis ein.

- **Programmiertrick: *Sentinel***
  - Deutsch: "Wächterwert"
  - Füge ans Ende des "linken" und "rechten" Arrays einen sonst nicht vorkommenden Wert maximalen hinzu (hier: ∞)
  - Verkürzt Code, <u>aber nur falls Integer zu sortieren sind.</u>




```
MERGE(A, l, m, r)          ← Verschmelze A[l..m] und A[l+1..r]
1    n1 = m – l + 1
2    n2 = r - m
3    let L[0..n1] and
          R[0..n2] be new arrays
4    for i = 0 to n1-1
5       L[i] = A[l+i]
6    for j = 0 to n2-1
7       R[j] = A[m+1+j]
8   L[n1] = ∞                  // Sentinel
9   R[n2] = ∞
10   i = 0
11   j = 0
12   for k = l to r            // Reißverschluss
13      if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16      else
17         A[k] = R[j]
18         j = j + 1
```

Quellcode: MergeSortRecursive.java

Publikums-Joker: Merge

Welche Aussage ist ***falsch***?

A. Die Funktion `MERGE` erfordert O(1) zusätzlichen Speicherplatz.

B. Die Funktion `MERGE` funktioniert nur, wenn die beiden Bereiche bereits sortiert sind.

C. Durch den Einsatz des Sentinels/Wächterwerts ∞ bzw. `Integer.MAX_VALUE` kann der Code von MERGE kurz gehalten werden.

D. Falls alle Werte des rechten Bereichs größer sind als alle Werte des linken Bereichs, nimmt `MERGE` keine Vertauschungen vor.

Diskussion

- **Laufzeit**
  - MERGE: $\Theta(n)$
    - „Es wird jedes Element der Eingabe ca. zweimal angeschaut.“
  - MERGE-SORT Rekursion, gesamt:
    - $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$
    - Geschlossene Form $T(n) = n \cdot \log n + n = \Theta(n \cdot log\, n)$ (siehe Maximum Subarray)

- **Speicherverbrauch**
  - MERGE benötigt $\Theta(n)$ zusätzlichen Speicherplatz.
    - Arrays *L* und *R*
  - ***Kein In-Place*** Algorithmus!
    - In-Place Algorithmus jedoch theoretisch möglich, wenn auch sehr kompliziert:
    - Ausblick: https://xinok.wordpress.com/2014/08/17/in-place-merge-sort-demystified-2/

- ***"Divide-Phase" einfach, Combine-Phase "schwierig"!***
  - Man kann beweisen, dass es asymptotisch keine schnelleren vergleichsbasierten Sortieralgorithmen gibt.

Mergesort: Iteration vs. Rekursion

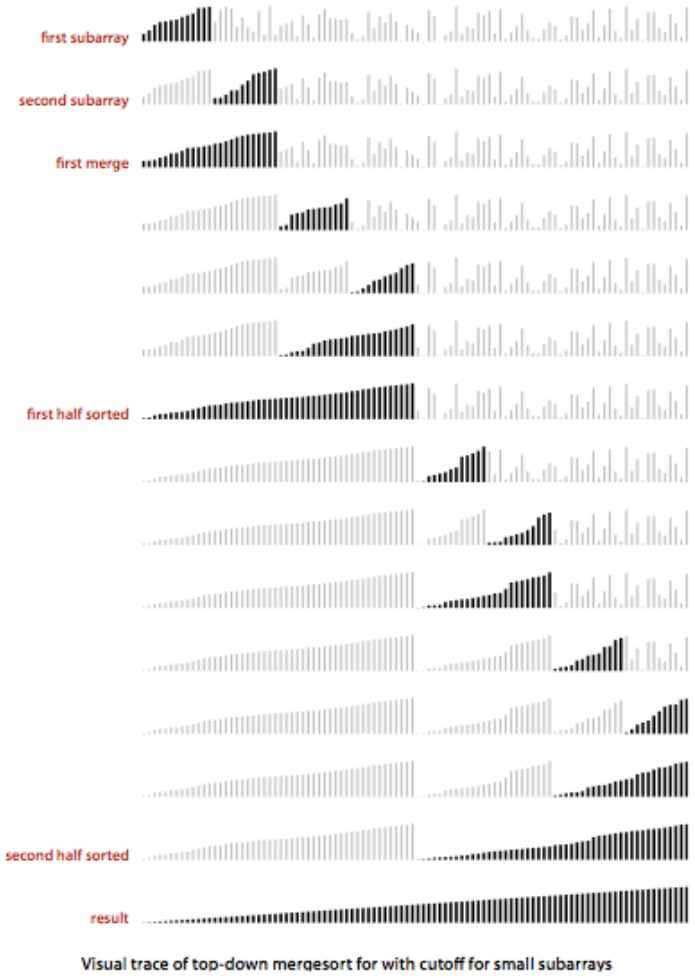
- Mergesort lässt sich ***iterativ*** oder ***rekursiv*** implementieren.
- **Animation**
  - https://algorithm-visualizer.org/divide-and-conquer/merge-sort
- ***Rekursiv: Top-Down Mergesort, Divide & Conquer***
  - Siehe bisherige Version, zunächst wird die linke Hälfte sortiert.
- ***Iterativ: Bottom-up***
  - Idee: Verschmelze erst 1-elementige, dann 2-elementige, dann 3-elementige Arrays.

```
MERGESORT-ITERATIVE(A)
1    n = a.length
2    let temp[0..(n-1)] be a new array   // create temp array for merging
4    for (len = 1; len < n; len *= 2)    // length of subarrays to merge
5       for (left = 0; left < n – len; left += 2 * len)
6          // iterate over left border of subarrays; always consider 2 subarrays
7          middle = left + len – 1
8          right = min{left+2*len-1, n-1}
9          merge(A, left, middle, right)
```

Quellcode: MergeSortIterative.java

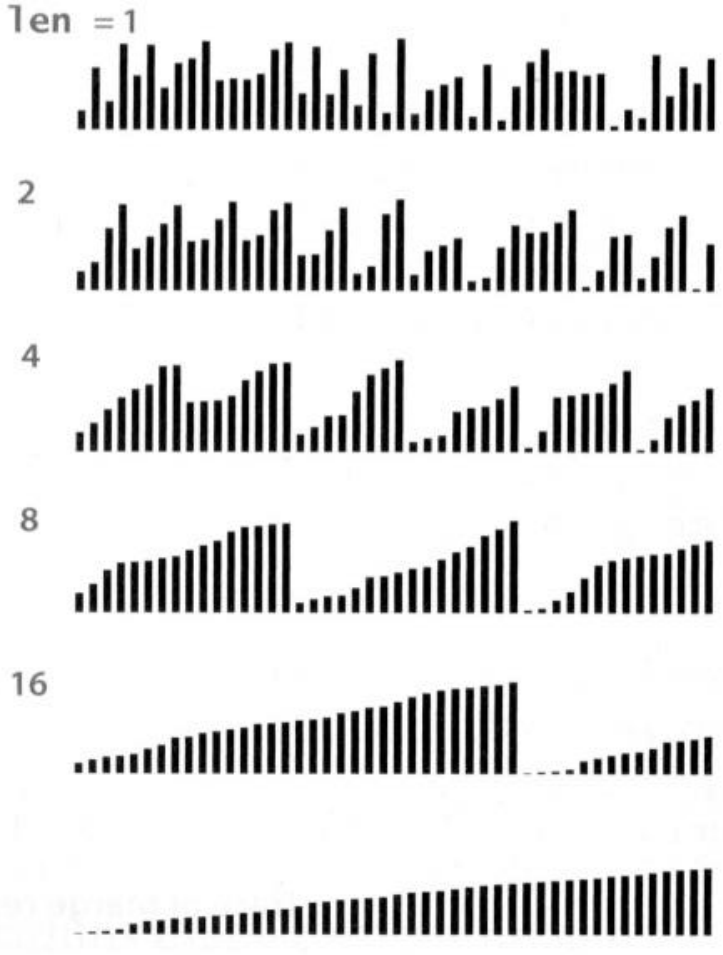
Visualisierung des Vorgehens

- Top-Down / rekursiv



Visual trace of top-down mergesort for with cutoff for small subarrays

- Bottom-Up / iterativ



Quelle: Sedgewick et al.

Publikums-Joker: Mergesort

Welche der folgenden Aussagen ist ***falsch***?

A. Mergesort ist für sehr große Eingaben schneller als Insertionsort.

B. Mergesort ist kein In-Place Algorithmus.

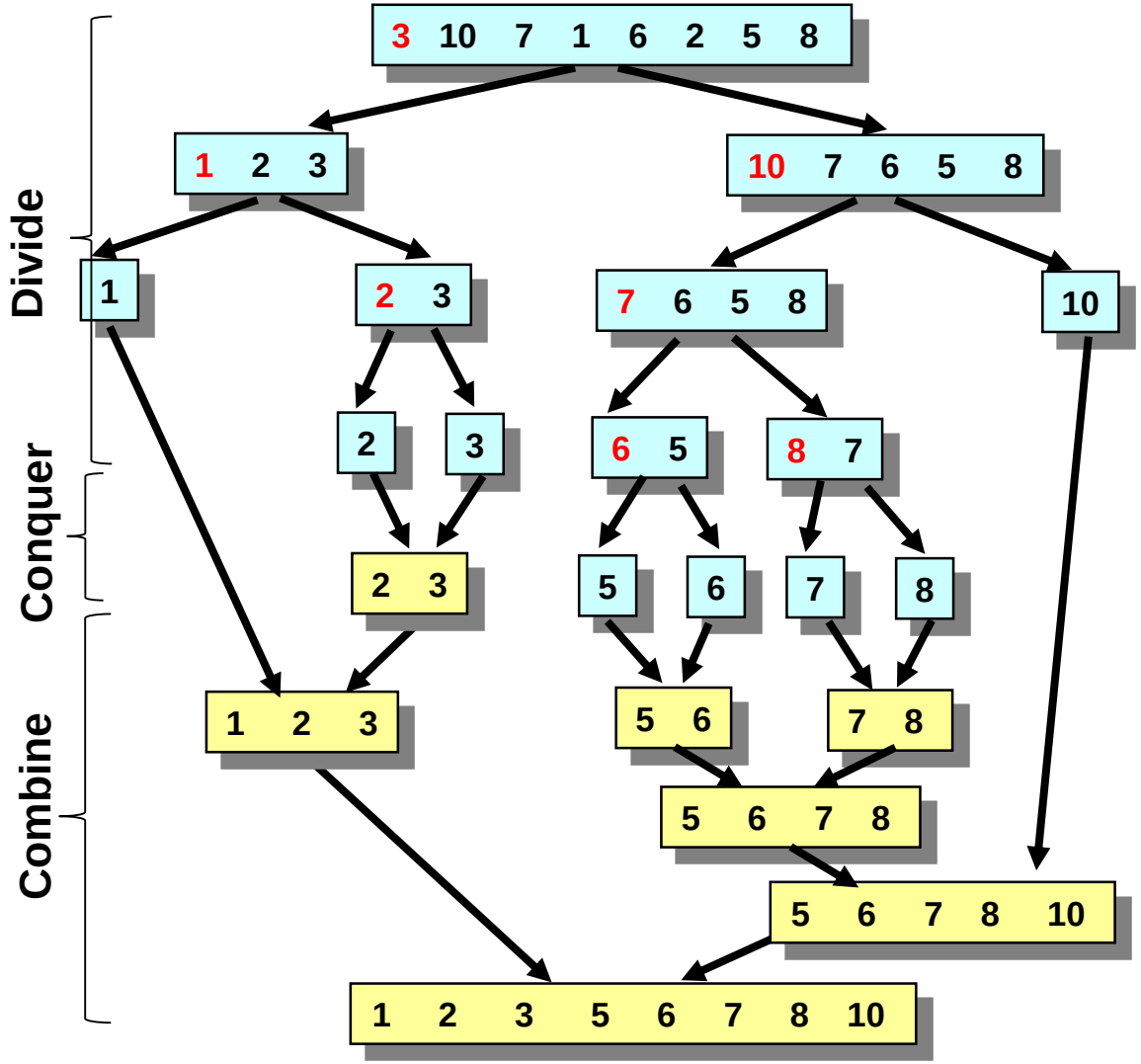
C. Wendet man Mergesort auf ein Array an, das nur 2 verschiedene Werte enthält, dann ist die Worst-Case Laufzeit $\Theta(n \log n)$.

D. Die iterative Variante ist bezüglich der asymptotischen Worst-Case Laufzeit schneller als die rekursive Variante.

Inhalt

- Einführung
- Mergesort
- **Quicksort [5]**
- Heapsort
- Sortieren in linearer Zeit
- Zusammenfassung

Sortieren mit Quicksort



Teilen durch Vergleichen

Sortieren beim Teilen

Zusammenfügen durch Vereinigen

Man wählt in jedem Schritt ein Element (**Pivot**) und partitioniert das Array bzgl. dieses Elements.

Divide & Conquer: Sortiere *A[l..r]*

- ***Pivot p:*** Bestimme beliebiges Element *A[p]* mit $l \leq p \leq r$
  - Strategie zunächst: Wähle Element ganz rechts, d.h. *A[r]*
- ***Divide:*** Partitioniere *A[l..r]* in *A[l..p-1]* und *A[p+1..r]*, so dass
  - jedes Element im 1. Subarray ≤ A[*p*].
  - jedes Element im 2. Subarray ≥ A[*p*].
  - Pivotelement *A[p]* bereits an ***korrekter, finaler Position*** steht
- ***Conquer:*** Sortiere rekursiv die beiden Teilarrays.
- ***Combine:*** Nichts zu tun!

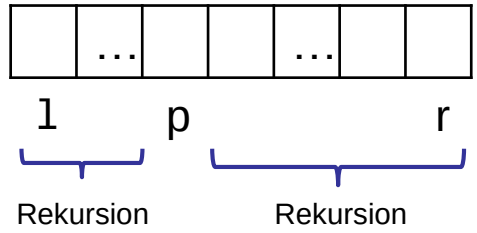
Wende Quicksort auf Subarray *A[l..r]* an

```
QUICKSORT(A, l, r)
1  if l < r
2     p = PARTITION(A, l, r)
3     QUICKSORT(A, l, p-1)
4     QUICKSORT(A, p+1, r)
```

Quellcode: QuickSort.java

PARTITION gibt Index p des Pivotelements nach Durchführung der Partitionierung zurück.

Rekursive Aufrufe



Partitionierung nach Hoare

- ***Herausforderungen:***
  - *In-Place:* Nur $O(1)$ zusätzlicher Speicher.
  - Pivot am Schluss an "richtiger" Stelle

- ***Partitionierung***
  - Wähle Pivot *x* am rechten Rand (Entscheidung willkürlich!)
  - Finde Tauschpartner
    - Zeiger *i* wandert von links nach rechts bis Element A[i] größer als Pivot.
    - Zeiger *j* wandert von rechts nach links bis Element A[j] kleiner als Pivot.
    - Dann *A[i]* und *A[j]* vertauschen.
  - Terminierung: Zeiger *i* und *j* kreuzen sich.
  - Eventuell noch Pivot an richtige Stelle setzen.

Partitioniere Array
A[l..r], so dass *A[l..p-1]* ≤ *A[p]* ≤ *A[p+1..r]*

```
PARTITION(A, l, r)
1   pivot = A[r]
2   i = l
3   j = r-1
4   do
5      while A[j]≥pivot and j>l
6         j = j-1
7      while A[i]≤pivot and i<r
8         i = i+1
9      if (i < j)
10        exchange(A[i], A[j])
11  while i<j
12  if A[i]> pivot,
13     exchange(A[i], A[r])
14  return i
```

Quellcode: QuickSort.java

Gibt Indexposition des Pivots zurück

Publikums-Joker: Partitionierung nach Hoare

Welche der folgenden Aussagen ist ***falsch***?

A. Steht rechts das größte Element des Arrays, so werden keinerlei Vertauschungen vorgenommen.

B. Falls alle Element des Arrays gleich sind, so nimmt `PARTITION` keinerlei Vertauschungen vor.

C. Die Laufzeit von PARTITION beträgt $\Theta(n)$.

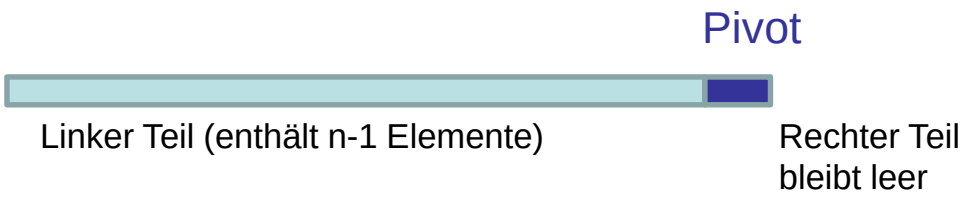
D. Es sei A=(2,1). Es wird PARTITION(A, 0, 1) aufgerufen. Dann wird nur eine Vertauschung vorgenommen und zwar in Zeile 10.

```
PARTITION(A, l, r)
1   pivot = A[r]
2   i = l
3   j = r-1
4   do
5      while A[j]≥pivot and j>l
6         j = j-1
7      while A[i]≤pivot and i<r
8         i = i+1
9      if (i < j)
10        exchange(A[i], A[j])
11  while i<j
12  if A[i]> pivot,
13     exchange(A[i], A[r])
14  return i
```

Quicksort: Laufzeit

- `PARTITION:` Lineare Laufzeit $\Theta(n)$
  - Jedes Element wird „einmal betrachtet“.

- **Worst Case**
  - Das Pivot-Element partitioniert das Ausgangsarray in zwei Arrays mit sehr ungleicher Größe.
    - Beispiel: Man wählt zufällig das größte Element als Pivotelement.
    - Wie viele Elemente fallen in den linken Teil? Wie viele in den rechten?
  - Rekursion: $T(n) = T(n-1) + \Theta(n) = \boldsymbol{\Theta(n^2)}$



- **Best Case**
  - Pivot teilt Ausgangsarray immer in genau 2 gleich große Hälften.
  - Rekursion: $T(n) = 2 \cdot T(n/2) + \Theta(n) = \boldsymbol{\Theta(n \log n)}$

Diskussion

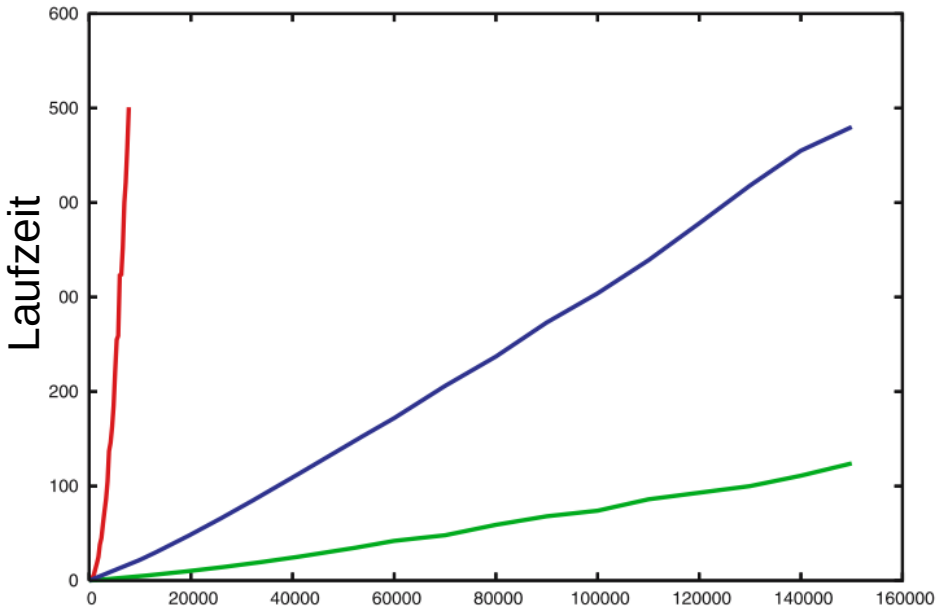
- **Laufzeit**
  - Worst Case: $\Theta(n^2)$
  - Average Case: $\Theta(n \log n)$
  - Die "Konstanten" in $\Theta(n \log n)$ sind im Vergleich zu Mergesort und Heapsort klein.

- **Speicherverbrauch**
  - In-Place!
  - Gut geeignet für virtuellen Speicher, räumliche Lokalität

- ***Variation:*** Randomisierter Quicksort-Algorithmus
  - Wähle Pivot zufällig und nicht immer das rechte Element
  - Dann wird der Worst Case unwahrscheinlicher.

- ***Divide-Phase schwierig, Combine-Phase leicht!***
  - Umgekehrt wie bei Mergesort.

Publikumsjoker



Eingabegröße

Algorithmen:
- Insertionsort
- Quicksort
- Mergesort

Experimentell bestimmte Laufzeiten in Millisekunden der drei Algorithmen zum Sortieren von Folgen der Länge 1 bis 150.000

Ordnen Sie die Farben den Algorithmen zu!

A. Rot=Insertion, Blau=Quick, Grün=Merge
B. Rot=Insertion, Blau=Merge, Grün=Quick
C. Rot=Merge, Blau=Insertion, Grün=Quick
D. Rot=Merge, Blau=Quick, Grün=Insertion
E. Rot=Quick, Blau=Merge, Grün=Insertion
F. Rot=Quick, Blau=Insertion, Grün=Merge

Inhalt

- Einführung
- Mergesort
- Quicksort
- **Heapsort**
- Sortieren in linearer Zeit
- Zusammenfassung

Heapsort

- ***Worst Case Laufzeit***: $O(n \log n)$
  - Vergleich: Quicksort hat im Worst Case $O(n^2)$
  - *Theorie*: Es kann kein schnelleres, allgemeines Sortierverfahren geben
  - Aber: Average Case bei Quicksort besser!

- Einsatz von Heapsort statt Quicksort lohnt sich nur wenn
  - Vergleiche auf zu sortierenden Daten sehr aufwendig sind und **gleichzeitig**
  - die Datenanordnung für Quicksort ungünstig ist.

- Heapsort ist In-Place!

- Benötigt eine ADT "**Heap**" (dt. "**Halde**")
  - Achtung: Mit "Heap" ist hier nicht wie bei Prg2 der Speicherbereich gemeint, der dynamische Daten aufnimmt und durch die Garbage Collection verwaltet wird.

Datenstruktur Heap („Halde“) und Heapsort

- Datenstruktur zur ***effizienten*** Bestimmung des ***maximalen*** bzw. ***minimalen*** Elements einer Menge!
  - Heapsort verwendet diese Datenstruktur.

- ***Vorgehen*** bei Heapsort / Überblick
  - Wiederholtes Entfernen des Maximums!

```
HEAPSORT
Verwandle unsortierte Folge/Array F in einen Heap
while (F nicht leer)
    - entnimm maximales Element aus Heap
    - setze maximales Element an korrekte Position
    - stelle Heapeigenschaft auf Rest wieder her
```

Maximum Heap

- **Definition *Max-Heap*:** Lineare Liste $(k_0, k_1, \ldots, k_{n-1})$, so dass für alle $i = 0, 1, \ldots, \frac{n-1}{2}$ gilt: $k_i \geq k_{2i+1}$ und $k_i \geq k_{2i+2}$ sofern $2i < n$ bzw. $2i + 1 < n$
  - ***Min-Heap*** Definition äquivalent
  - Fast immer wird ein Array zur Umsetzung der Linearen Liste bzw. des Heaps verwendet.

- **Übung**: Erfüllen diese Folgen die Heap-Eigenschaft?
  -

i = 0	*i = 1*	*i = 2*	*i = 3*	*i = 4*	*i = 5*	*i = 6*	*i = 7*	*i = 8*	*i = 9*
16	14	10	4	7	9	3	2	8	1

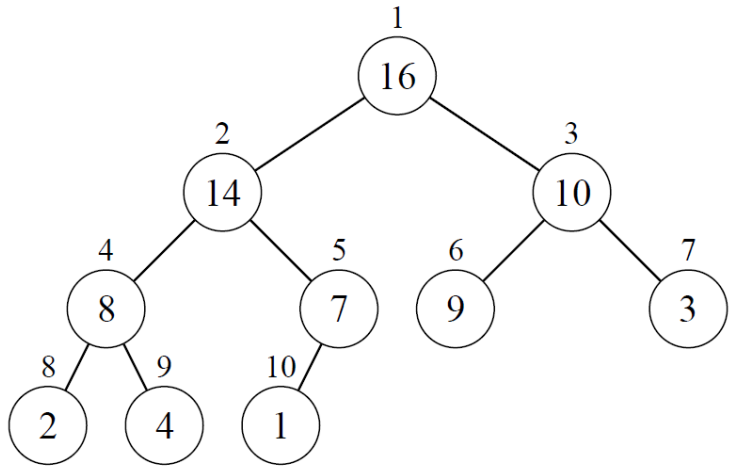
i = 0	*i = 1*	*i = 2*	*i = 3*	*i = 4*	*i = 5*	*i = 6*
47	17	43	15	8	4	2

Visualisierung eines Heaps als Baum

- Ein Heap
  - Wird meist in Form eines Arrays abgespeichert,
  - Kann jedoch besser als Baum ***graphisch visualisiert*** werden.

- Nur auf den ersten Blick ein binärer Suchbaum!
  - Es gibt nur eine schwache Ordnung zwischen den Elementen.

- **Heap-Bedingung**
  - ***Max-Heap***: Schlüssel ***jedes*** Knotens ≥ Schlüssel seiner beiden Kinder.
  - ***Min-Heap:*** Schlüssel ***jedes*** Knotens ≤ Schlüssel seiner beiden Kinder.
  - Darauf folgt: Die Wurzel speichert den größten bzw. kleinsten Wert.



Quelle: [1]

Achtung: Dieses Array ist 1-indiziert!

Publikums-Joker: Heap

Ein Array enthalte genau 4 verschiedene Elemente. Wie viele Möglichkeiten für die Belegung dieses Arrays gibt es?

A. 1

B. 2

C. 3

D. 4

Heap: Operationen und Navigation

- Navigation
  - Man kann zwischen Eltern und Kindknoten im Array durch Indexrechnung navigieren.
  - **`PARENT(i):`** Index des Elternknotens von *i*
    - `return (i – 1) : 2`
  - **`LEFT(i):`** Index des linken Kindknotens von *i*
    - `return 2 * i + 1;`
  - **`RIGHT(i):`** Index des rechten Kindknotens von *i*
    - Wie berechnet man diesen Index?

- **`BUILD-MAX-HEAP(A):`** Operation
  - Baut aus beliebigem (0-inidiziertem) Array *A* der Größe *n* ein Array, dass der MaxHeap-Eigenschaft genügt.

- **`MAX-HEAPIFY (A, i, n):`** Operation
  - Berücksichtigt nur den Heap *A[0..(n-1)].* Alle Elemente weiter rechts werden ***ignoriert***.
  - Stellt Heap-Bedingung für den Unterbaum ab Index *i* wieder her, falls diese verletzt ist.

MAX-HEAPIFY: Stelle Heap-Bedingung her

```
HEAPSORT
Verwandle unsortiertes Array F in einen Heap
while (F nicht leer)
    entnehme maximales Element aus Heap
    setze maximales Element an korrekte Position
    stelle Heapeigenschaft auf Rest wieder her
```

- **Idee**
  - Maximum (= Element ganz links im Array, kleinster Index)
  - Tausche Maximum mit Element ganz rechts. Das Maximum steht dann an korrekter Stelle.
  - Aber nun ist die Heapeigenschaft verletzt, da nicht zwingend das größte Element an der Heapwurzel steht.
  - Rufe `MAX-HEAPIFY` auf der „neuen“ Wurzel auf, um die Heapeigenschaft dort wiederherzustellen

Wiederherstellen der Heap-Eigenschaft

- **Heap-Bedingung sei an Knoten *i* verletzt**
  - Schlüssel des Elternknotens ist kleiner als Schlüssel eines seiner beiden Kinder.
- Vertausche Elternknoten mit ***größerem*** der beiden Kinder
  - "Versickern"
- Nun kann Heap-Bedingung weiter unten verletzt sein
  - Rekursion!

Stelle Heap-Eigenschaft im Teilbaum mit A[i] als Wurzel wieder her. Aber nur bis zum Index n-1!

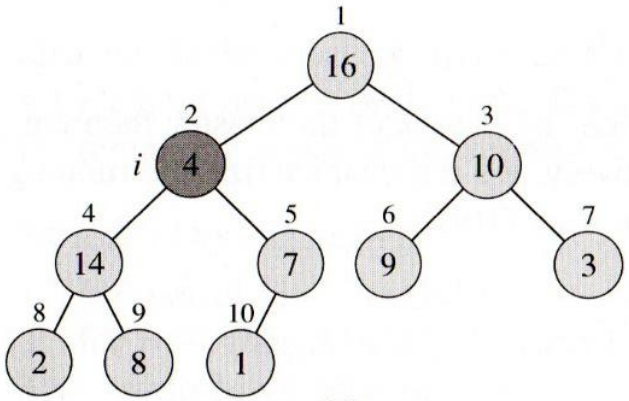
```
MAX-HEAPIFY(A, i, n)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l < n and A[l] > A[i]
4      largest = l
5   else
6      largest = i
7
8   if r < n and A[r] > A[largest]
9      largest = r
10
11  if largest ≠ i
12     exchange(A[i],A[largest])
13     MAX-HEAPIFY(A, largest, n)
```

Rekursion!

Quellcode: HeapSort.java

Wiederherstellen der Heap-Eigenschaft

- Übung: Stellen Sie die Heap-Eigenschaft wieder her!

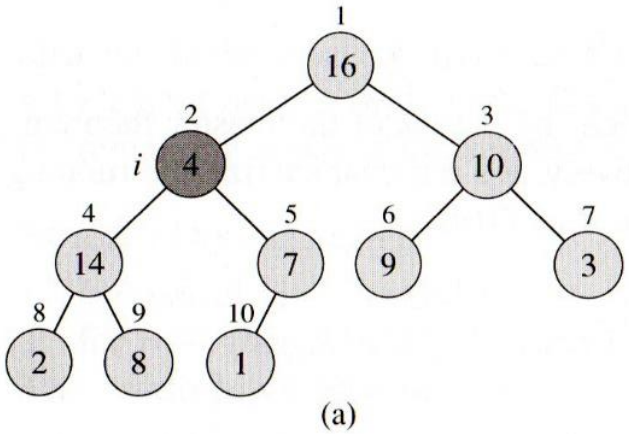


Quelle: [1]

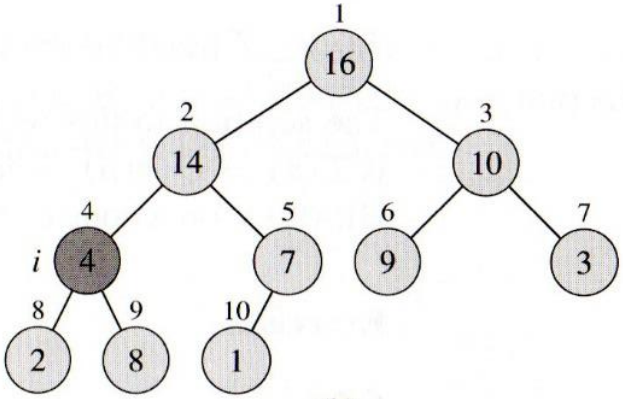
Achtung: Arrays sind hier 1-indiziert!

Wiederherstellen der Heap-Eigenschaft

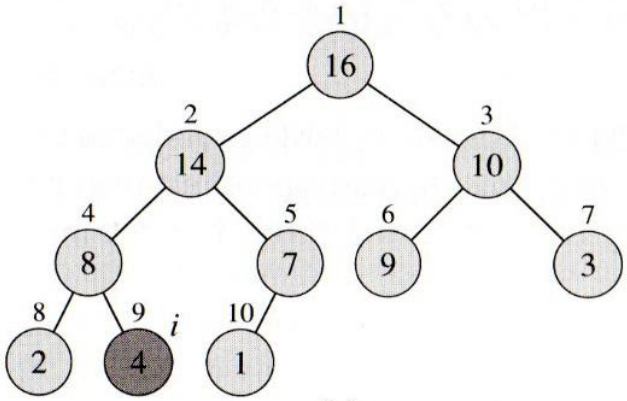
- Lösung



(a)



(b)



(c)

Quelle: [1]

Achtung: Arrays sind hier 1-indiziert!

Initiales Herstellen der Heap-Eigenschaft

```
HEAPSORT
Verwandle unsortierte Folge/Array F in einen Heap
while (F nicht leer)
    entnehme maximales Element aus Heap
    setze maximales Element an korrekte Position
    stelle Heapeigenschaft auf Rest wieder her
```

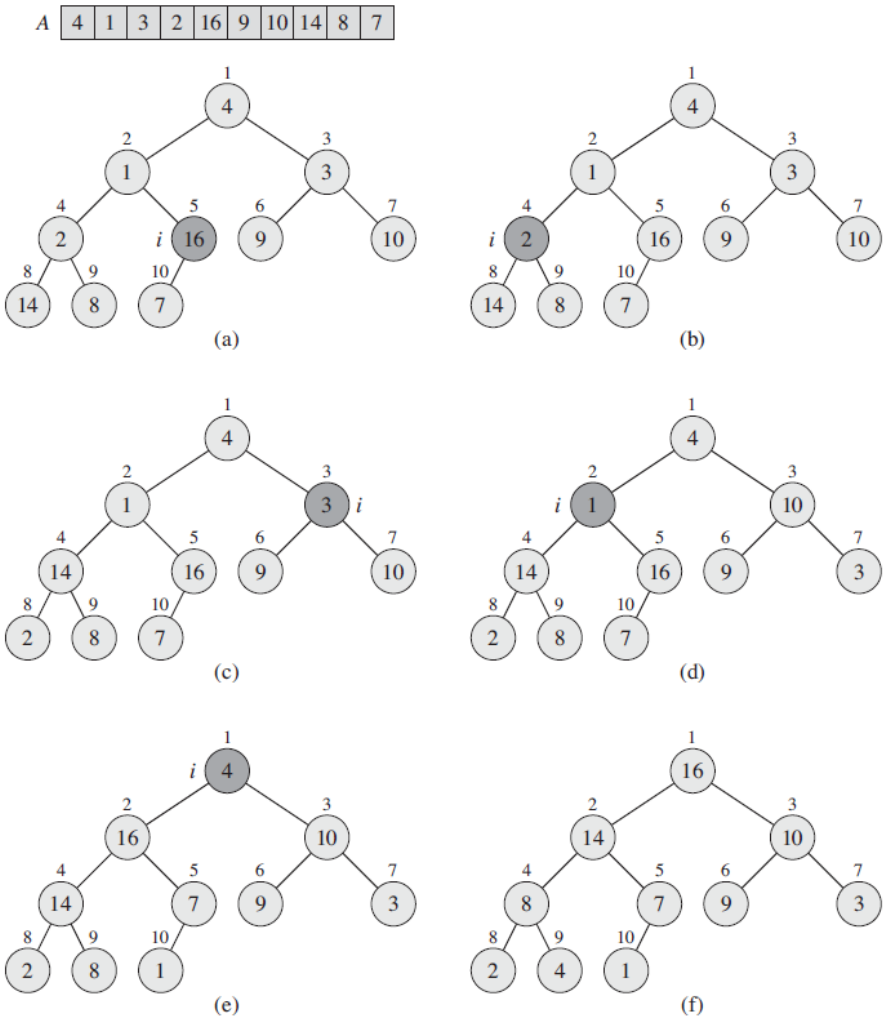
- Array zu Beginn kein Heap → Heap herstellen!
- **Idee**
  - ***Alle Blätter*** (= Elemente A[*n*/2..(n-1)]) erfüllen trivialerweise bereits die Heap-Bedingung
  - Rufe `MAX-HEAPIFY` auf ***allen verbleibenden Elementen*** auf und zwar in der folgenden Index-Reihenfolge: (n-1)/2, (n-1)/2 - 1, … ,0

```
BUILD-MAX-HEAP(A)
1   n = A.length
2   for i = (n − 1)/2 downto 0
3      do MAX-HEAPIFY(A, i, n)
```

Wandle das Array A in einen Heap um

Quellcode: HeapSort.java

BUILD-MAX-HEAP: Beispiel



Achtung: In der Abbildung ist das Array 1-indiziert!

```
BUILD-MAX-HEAP(A)
1   n = A.length
2   for i =(n − 1)/2 downto 0
3      do MAX-HEAPIFY(A, i, n)
```

a) Anfangszustand
b)- e) Zwischenzustände
f) Endzustand

Quelle: [1]

Heapsort Algorithmus: Überblick

```
HEAPSORT(A)
1    BUILD-MAX-HEAP(A)
2    for i = n-1 downto 1
3        exchange(A[0],A[i])
4        MAX-HEAPIFY(A, 0, i-1)
```

```
HEAPSORT
Verwandle unsortiertes Array F in einen Heap
while (F nicht leer)
    entnehme maximales Element aus Heap
    setze maximales Element an korrekte Position
    stelle Heapeigenschaft auf Rest wieder her
```

Tausche Wurzel mit Element A[i] (=Element rechts unten);

```
BUILD-MAX-HEAP(A)
1    n = A.length
2    for i = (n − 1)/2 downto 0
3        do MAX-HEAPIFY(A, i, n)
```

```
MAX-HEAPIFY(A, i, n)
1    l = LEFT(i)
2    r = RIGHT(i)
3    if l < n and A[l]>A[i]
4        largest = l
5    else
6        largest = i
7
8    if r < n and A[r] > A[largest]
9        largest = r
10
11   if largest ≠ i
12       exchange(A[i],A[largest])
13       MAX-HEAPIFY(A, largest, n)
```

Quellcode: HeapSort.java

Worst Case Laufzeit: “intuitiv”

- **`MAX-HEAPIFY (A,i,n)`**
  - Vertausche ggfs. Eltern mit größerem Kind: O(1)
  - Nun muss rekursiv an Unterbäumen Heap-Bedingung wiederhergestellt werden
  - Baum der Höhe *h* hat mindestens 2^h Elemente → h ≈ log n
  - h Rekursionen → Laufzeit: $\boldsymbol{\Theta(\log n)}$

- **`BUILD-MAX-HEAP(A)`**
  - Ruft *n*/2- mal `MAX-HEAPIFY` auf
  - Gesamtlaufzeit: *n*/2 * log n → $\boldsymbol{\Theta(\mathrm{n}\, log\, n)}$

Heapsort Algorithmus: Laufzeit

```
HEAPSORT(A)
1    BUILD-MAX-HEAP(A)
2    for i = n-1 downto 1
3        exchange(A[0],A[i])
4        MAX-HEAPIFY(A, 0, i-1)
```

$\Theta(n)$ *mal* MAX-HEAPIFY $\rightarrow \Theta(n \log n)$

```
BUILD-MAX-HEAP(A)
1    n = A.length
2    for i = ⌊n/2⌋ downto 0
3        do MAX-HEAPIFY(A, i, n)
```

$\Theta(n \log n)$

$\Theta(\log n)$

```
MAX-HEAPIFY(A, i, n)
1    l = LEFT(i)
2    r = RIGHT(i)
3    if l ≤ n and A[l]>A[i]
4        largest = l
5    else
6        largest = i
7
8    if r ≤ n and A[r] > A[largest]
9        largest = r
10
11   if largest ≠ i
12       exchange(A[i],A[largest])
13       MAX-HEAPIFY(A, largest, n)
```

Gesamtlaufzeit: $\Theta(n \log n)$

Diskussion

- **Laufzeit**
  - Worst Case: $\Theta(n \log n)$
  - Average Case: $\Theta(n \log n)$
  - Bzgl. des asymptotischen Verhaltens kann es kein besseres vergleichsbasiertes Sortierverfahren.
    - Vergleichsbasiert == Beruhend auf dem Vergleich von Schlüsseln.

- **Dennoch:**
  - Ein gut implementierter Quicksort ist meist schneller!

- **Speicherverbrauch**
  - In-Place, kein zusätzlicher Speicher notwendig!

- **Animation**
  - https://www.cs.usfca.edu/~galles/visualization/HeapSort.html
  - https://algorithm-visualizer.org/brute-force/heapsort

Publikums-Joker: Heapsort

Welche der folgenden Aussagen ist ***falsch***?

A. Heapsort basiert auf einem Min-Heap.

B. Heapsort läuft nie langsamer als $\Theta(n \log n)$.

C. Heapsort entfernt jeweils das Maximum aus dem Heap und tauscht es mit dem rechtesten Element des noch unsortierten Bereichs.

D. Heapsort kann ungünstig sein, da teilweise zwischen Speicheradressen hin- und hergesprungen wird.

Inhalt

- Einführung
- Mergesort
- Quicksort
- Heapsort
- **Sortieren in linearer Zeit**
- Zusammenfassung

Vergleichsbasierte Sortierverfahren

- Alle bisherigen Suchverfahren sind **vergleichsbasiert.**
  - Es wird jeweils ein Paar von Schlüssel miteinander vergleichen.

- Theorie: $\Omega(n \log n)$ ist eine **untere Schranke** für die (Worst Case)-Laufzeit von vergleichsbasierten Verfahren.
  - Es kann kein schnelleres Sortierverfahren geben, so lange man mit Vergleichen von Schlüsseln arbeitet.

- Macht man weitere **Annahmen über die Schlüssel**, so sind jedoch schnellere Verfahren möglich
  - ***Countingsort***
  - ***Radixsort***

CountingSort

- **Annahme** bzgl. Schlüssel
  - Der Wertebereich der zu sortierenden Schüssel ist ***klein*** und ***bekannt.***
  - Beispiel im Folgenden: Integer aus der Menge $\{0, 1, \ldots, k\}$
    - ***k+1*** verschiedene Werte.
    - Maximaler Schlüsselwert: ***k***.

- **Grundidee**
  - Berechne für jeden Schlüssel *x* die Anzahl der Elemente, die kleiner sind als *x*.
  - Beispiel: Falls 17 Elemente kleiner sind als *x*, dann muss *x* an der 18. Position des Arrays stehen.

Sortiere Array A mit n Elementen, wobei die zu sortierenden Zahlen im Bereich {0, 1, 2, …k} liegen.

COUNTINGSORT(A,*n*,*k*) *(Skizze)*
Eingabe: n-elementiges Array mit $A[j] \epsilon \{0,1,..,k\}$ für $j = 1, \ldots, n$
Ausgabe: n-elementiges Array mit Werten aus A, **sortiert!!!**.
Zwischenspeicher: *C*[0..*k*]
1. Zähle zunächst in *C*[*i*] wie oft jeder Wert **i** $\epsilon \{0,1,..,k\}$ vorkommt.
2. Berechne dann in *C*[*i*] wie viele Werte $\leq i$ sind.
3. Gehe von hinten durch A, setze jeden Wert an korrekte Position in *B*.

Countingsort: Beispiel

Sortiere Array A mit n Elementen, wobei die zu sortierenden Zahlen im Bereich {0, 1, 2, …k} liegen.

COUNTINGSORT(A,*n*,*k*) *(Skizze)*
Eingabe: n-elementiges Array mit $A[j] \epsilon \{0,1,..,k\}$ für $j = 1,\ldots,n$
Ausgabe: n-elementiges Array mit Werten aus A, **sortiert!!!**.
Zwischenspeicher: *C*[0..*k*]
1. Zähle zunächst in *C*[*i*] wie oft jeder Wert **i** $\epsilon \{0,1,..,k\}$ vorkommt.
2. Berechne dann in *C*[*i*] wie viele Werte $\leq i$ sind.
3. Gehe von hinten durch A, setze jeden Wert an korrekte Position in *B*.

Beispiel: siehe Übung

- Beispiel: A=[2, 5, 3, 0, 2, 3, 0, 3}
  - Wie sieht C nach Schritt 1 aus?
    - An Index i+1 wird gezählt wie oft Zahl i vorkommt.
  - Wie sieht C nach Schritt 2 aus?
    - An C[i] kann man ablesen wie viele Elemente in der Eingabe ≤ *i* sind. **Folglich kann man das Eingabeelement i an die i. Position im Ergebnisarray setzen.**
  - Beim Durchlaufen von A: Wie sieht A nach jeder Iteration aus?

- Animation:
  - https://www.cs.usfca.edu/~galles/visualization/CountingSort.html

Stabiler Sortieralgorithmus

- ***Stabil vs. instabil***
  - Kommen in Eingabe Schlüssel mehrfach vor, so behält der Algorithmus die Reihenfolge dieser Elemente bei.
  - Beispiel: $A=<1_a, 2, 1_b>$ *wird sortiert.*
    - Stabiles Verfahren liefert: $<1_a, 1_b, 2>$
    - Instabiles Verfahren könnte liefern: $<1_b, 1_a, 2>$
  - Stabiles Verfahren sind manchmal notwendig → wesentlich für RadixSort!

- Animation: Stabiler Countingsort
  - https://www.cs.usfca.edu/~galles/visualization/CountingSort.html

- Verzichtet man auf Stabilität lässt sich Countingsort noch einfacher implementieren:
  - https://visualgo.net/bn/sorting

Countingsort: Diskussion

- Verwendet keine Vergleiche
  - Aber Annahme: Wertebereich der Schüssel ist ***klein*** und ***bekannt.***
  - Schlüssel können Integerzahlen sein.
  - Alternativ auch: Character aus einem Alphabet, siehe Übung.

- **Laufzeit**
  - $\Theta(n+k)$ (k: maximaler Integerwert)
  - Falls $k = O(n)$ wird die Laufzeit linear: $\Theta(n)$
  - Countingsort lohnt sich, falls maximaler Integerwert nicht zu groß bzw. deutlich kleiner als die Größe des Eingabearrays.

- CountingSort wird in der Regel in der ***stabilen*** Variante implementiert.

- Ist Countingsort ein **In-Place** Algorithmus?
  - Nein, ggfs. **sehr viel zusätzlicher** Speicher notwendig für *B* und *C*.
  - Radixsort ist eine Erweiterung von Countingsort und benötigt weniger zusätzlichen Speicher!

Publikums-Joker: Countingsort

Welche der folgenden Aussagen ist ***falsch***?

A. Countingsort benötigt O(n) zusätzlichen Speicherplatz.

B. Countingsort eignet sich für das Sortieren von Matrikelnummern einer Fachhochschule.

C. Countingsort ist ein stabiler Sortieralgorithmus, der die Reihenfolge von gleichen Eingabewerten nicht verändert.

D. Countingsort eignet sich für das Sortieren von Studenten nach Ihrer Klausurnote im Fach AD.

(LSD)-Radixsort

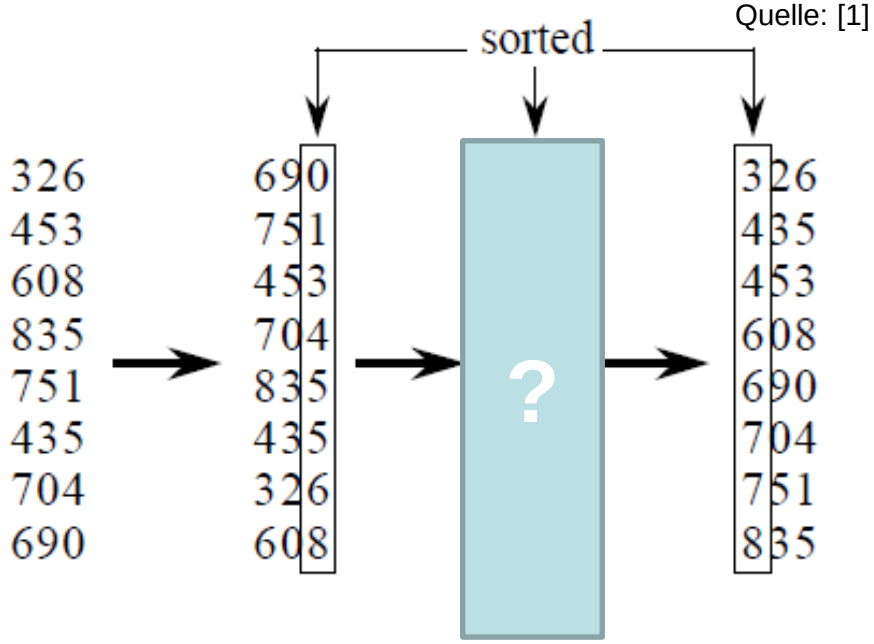
- **Idee**
  - Betrachte die Ziffern einer Zahl (z.B. Dezimalzahl) der Reihe nach
  - Sortiere erst nach niedrigstwertigster Ziffer, dann zweitniedrigstwertigster Ziffer, etc.
    - Meist mit Countingsort.
  - Intuitiver Ansatz erst nach der höchstwertigen Ziffer zu sortieren, würde viel zusätzlichen Speicherplatz (!) benötigen.

- Um eine Zahl mit ***d Ziffern*** zu sortieren.

```
RADIXSORT(A, d)
1    for i = 1 to d
2        use stable sort to sort A on
          ith least significant digit
```

Sortiere erst niedrigwertige, dann höherwertige Ziffern!

Quelle: [1]



Analyse

```
RADIXSORT(A, d)
1   for i = 1 to d
2      use a stable sort to
       sort A on  ith
       least significant digit
```

$\Theta(d)$ *mal*

Countingsort

Beispiel: siehe Übung

n: Größe des Eingabearrays
d: Anzahl Ziffern
k: Maximale Ziffer, bzw. wie viele verschiedene Ziffern gibt es?

- Verwende Countingsort als stabilen Sortieralgorithmus!
- In-Place?
  - CountingSort benötigt zusätzlichen Speicher. → Kein In-Place!
  - Aber ansonsten kein weiterer Speicher notwendig, falls man wie bei Radixsort mit der niedrigstwertigsten Ziffer beginnt.
- Laufzeit
  - $\Theta(d(n+k))$
  - Falls $k = O(n)$, d.h. maximaler Wert ähnlich wie Eingabegröße:

Exkurs: Generische Anwendung von Radixsort

n: Größe des Eingabearrays
d: Anzahl Ziffern / Stellen
k: Maximale Ziffer, bzw. wie viele verschiedene Ziffern gibt es?

- **„Wie teilt man einen Schlüssel in Ziffern auf“?**
  - Eine „Runde“ in Radixsort muss nicht zwingend einer Ziffer entsprechen!
  - Z.B. Gruppen von Ziffern könnten auf einmal mit Countingsort sortiert werden.
  - Oder: Schlüssel setzt sich aus mehreren Komponenten zusammen. Bsp: Für „Name“, „Alter“, „Geburtsdatum“ jeweils eine Runde mit Countingsort?

- **Allgemein**
  - Ein Schlüssel bestehe aus *b* Bits.
  - Unterteile Schlüssel in Gruppen von r Bits. Dann: $d = \lceil b/r \rceil$
  - Verwende jeweils Countingsort mit $k = 2^r - 1$
  - Laufzeit: $\Theta(\frac{b}{r} \cdot (n + 2^r))$
  - ***Theorie:*** Bei der idealen Wahl von $r := \log n$ erhält man $\Theta(bn/\log n)$
  - Kann deutlich besser sein, als Quicksort oder Mergesort.

Publikums-Joker: Radixsort

Radixsort wird ziffernweise auf die Folge

21, 86, 124, 33, 29, 163 angewendet. Welche Ordnung haben die Zahlen bevor in der letzten Runde die größte Ziffer betrachtet wird?

A. 21,33,163,124,86,29.
B. 21,29,33,86,124,163.
C. 21,124,29,33,163,86.
D. 21,29,86,33,124,163

Inhalt

- Einführung
- Mergesort
- Quicksort
- Heapsort
- Sortieren in linearer Zeit
- **Zusammenfassung**

Sortieren in Java

- `Java.util.Arrays`
  - `sort(int[] a)`: Verwendet effizienten Quicksort.
  - `sort(Object[] a):` Verwendet effizienten Mergesort.

- `Java.util.Collections`
  - `sort(.):` Verwendet effizienten Mergesort.

- Für andere Sortierverfahren müssen externe Frameworks verwendet werden.
  - Z.B. http://psjava.org/

Sortierverfahren: Vergleich

Algorithmus	Average Case	Worst Case	In-Place	Vergleichsbasiert
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	Ja	Ja
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	Ja	Ja
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	Nein	Ja
Quicksort	$\Theta(n \log n)$	$\Theta(n^2)$	Ja	Ja
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	Ja	Ja
Countingsort	Für bestimmte Eingaben linear	$\Theta(n + k)$	Nein	Nein
Radixsort	Für bestimmte Eingaben linear	$\Theta(d(n + k))$	Nein	Nein

n: Anzahl der Elemente
k: Anzahl der möglichen Werte, Ziffern
d: Anzahl der Stellen /Digits

Viele weitere Sortierverfahren!
https://de.wikipedia.org/wiki/Sortierverfahren

Inhalt

- Einführung und elementare Sortierverfahren
  - Bubblesort, Insertionsort, quadratische Laufzeit.
- Mergesort
  - Divide-Phase leicht, Combine-Phase schwierig.
- Quicksort
  - Divide-Phase schwierig, Combine-Phase leicht.
- Heapsort
  - Benötigt Datenstruktur Heap
  - Asymptotisch effizient, aber meist ist gut implementierter Quicksort schneller
- Sortieren in linearer Zeit
  - Macht man Annahmen über Schlüssel, so sind auch schnellere Verfahren möglich.
- Zusammenfassung

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 1.2.3, 5. Auflage, Spektrum Akademischer Verlag, 2012. (xxx)

[3] http://dilbert.com/strip/1996-05-22 (abgerufen am 21.10.2016)

[4] https://s3.amazonaws.com/lowres.cartoonstock.com/business-commerce-pc-neat-tidy_desks-desks-file-cwln557_low.jpg (abgerufen am 16.10.2016)

[5] https://de.wikipedia.org/wiki/Quicksort (abgerufen am 25.10.2017)