



Algorithmen und Datenstrukturen

Sortieren

Matthias Teschner
Graphische Datenverarbeitung
Institut für Informatik
Universität Freiburg

SS 09

Lernziele der Vorlesung



- Algorithmen
 - Sortieren, Suchen, Optimieren
- Datenstrukturen
 - Repräsentation von Daten
 - Listen, Stapel, Schlangen, Bäume
- Techniken zum Entwurf von Algorithmen
 - Algorithmenmuster
 - Greedy, Backtracking, Divide-and-Conquer
- Analyse von Algorithmen
 - Korrektheit, Effizienz

Überblick



- Einführung
- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Heap-Sort
- Quick-Sort
- Merge-Sort
- Counting-Sort

Sortierproblem



- **Eingabe:** Folge von Zahlen $\langle a_1, a_2, \dots, a_n \rangle$
- **Ausgabe:** Sortierte Folge der Eingabe $\langle a'_1, a'_2, \dots, a'_n \rangle$
mit $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- Eingabemenge als Feld oder verkettete Liste repräsentiert
- Sortierverfahren lösen das durch die Eingabe-Ausgabe-Relation beschriebene Sortierproblem.

Struktur der Daten



- zu sortierende Werte (Schlüssel) sind selten isoliert, sondern Teil einer größeren Datenmenge (Datensatz, Record)
- Datensatz besteht aus Schlüssel und Satellitendaten.
- Satellitendaten werden mit Schlüssel umsortiert.
- Im Folgenden werden Satellitendaten ignoriert. Konzepte werden immer lediglich für Schlüssel erläutert.
- Allgemeine Sortierverfahren basieren auf Schlüsselvergleich und ändern ggf. die Reihenfolge der Datensätze.

Sortieren - Bedeutung



- sehr fundamentales Problem
- als eigenständiges Problem
 - Notenlisten (nach Note oder nach Matrikelnummer)
- als Unterroutine
 - als Vorbereitung für Sweep-Algorithmen
- beweisbare untere Schranke
 - kann zur Analyse anderer Algorithmen verwendet werden

Strategien - Einführung



- Sortieren von Spielkarten
- Bubble Sort
 - Aufnehmen aller Karten vom Tisch
 - vertausche ggf. benachbarte Karten, bis Reihenfolge korrekt
- Selection Sort
 - Aufnehmen der jeweils niedrigsten Karte vom Tisch
 - Anfügen der Karte am Ende
- Insertion Sort
 - Aufnehmen einer beliebigen Karte
 - Einfügen der Karte an der korrekten Position
- optimales Verfahren meist nicht ausschließlich durch die mittlere Laufzeit bestimmt, sondern z. B. auch durch die Beschaffenheit der Eingabemenge

Eigenschaften

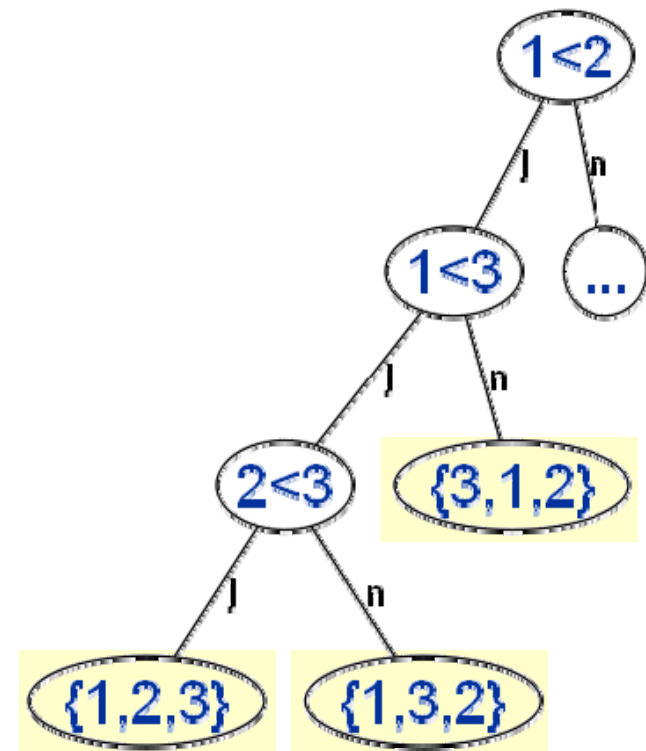


- **Effizienz**
 - Best-, Average-, Worst-Case
- **Speicherbedarf**
 - in-place (zusätzlicher Speicher von der Eingabegröße unabhängig)
 - out-of-place (Speichermehrbedarf von Eingabegröße abhängig)
- **rekursiv oder iterativ**
- **Stabilität**
 - stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht (wichtig bei mehrfacher Sortierung nach verschiedenen Schlüsseln)
- **verwendete Operationen**
 - Vertauschen, Auswählen, Einfügen
- **Verwendung spezieller Datenstrukturen**

Untere Schranke für vergleichsbasiertes Sortieren



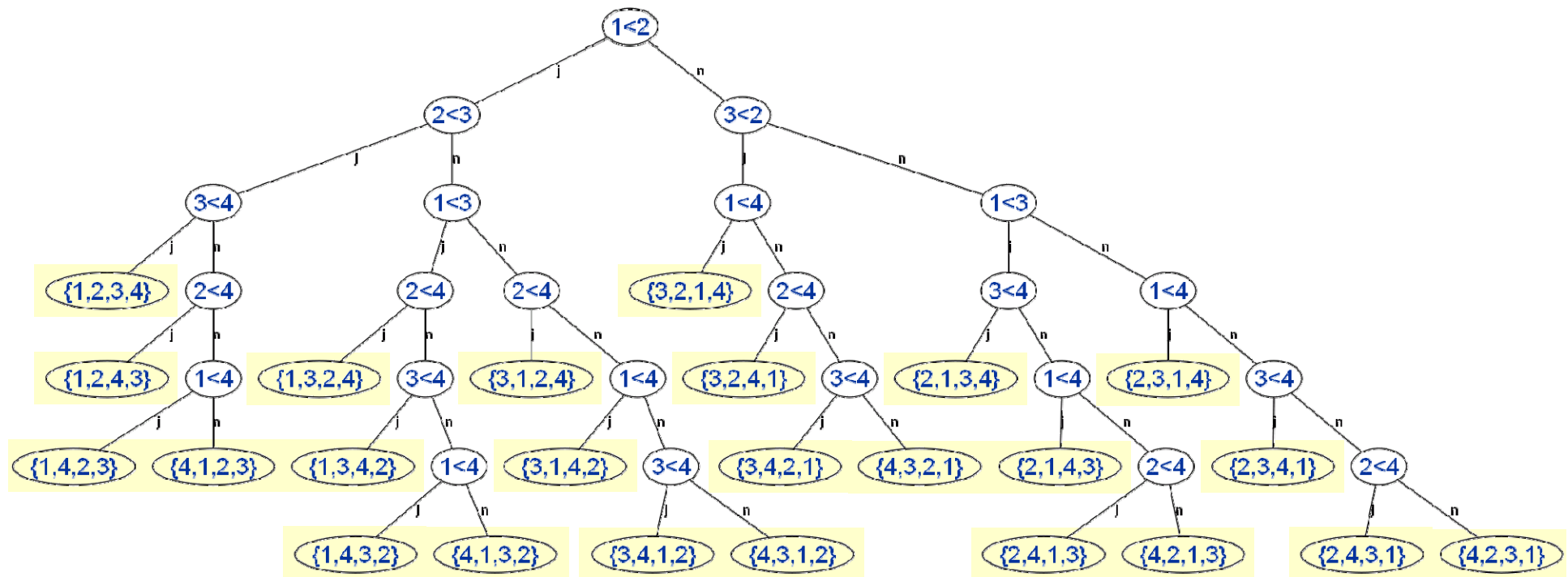
- Schlüsselvergleiche und Elementvertauschungen
- Ausgabe entspricht einer aus $n!$ Permutationen der Eingabe
- Entscheidungsbaum
 - Knoten liefern Informationen, um die in Frage kommenden Permutationen in zwei Teile zu zerlegen
 - Blätter enthalten Permutation, die alle vorhergehenden Relationen erfüllen
 - z. B. $\{1,2,3\}$



Entscheidungsbaum



- 24 mögliche Lösungen für $\{1,2,3,4\}$



- 24 ($n!$) Blätter eines binären Baums

Abschätzung der Tiefe des Entscheidungsbaums



- in Entscheidungsbaum mit $n!$ Blättern ist die mittlere und die maximale Tiefe eines Blattes bestenfalls $\log n!$

$$\begin{aligned}\log n! &= \log (1 \cdot 2 \cdot \dots \cdot n - 1 \cdot n) \\ &= \log (1 \cdot 2 \cdot \dots \cdot \frac{n-1}{2} \cdot \frac{n}{2} \cdot \dots \cdot n - 1 \cdot n) \\ &\geq \log \left(\frac{n}{2} \cdot \dots \cdot n - 1 \cdot n \right) \\ &\geq \log \left(\frac{n}{2} \right)^{\frac{n}{2}} \\ &= \frac{n}{2} \log (n) - 1 \\ &\in \Omega(n \log n)\end{aligned}$$

- untere Schranke für vergleichsbasiertes Sortieren
 $\Omega(n \log n)$

Überblick



- Einführung
- **Bubble-Sort**
- Selection-Sort
- Insertion-Sort
- Heap-Sort
- Quick-Sort
- Merge-Sort
- Counting-Sort

Prinzip



- durchlaufe die Menge
- vertausche zwei aufeinanderfolgende Elemente, wenn ihre Reihenfolge nicht stimmt
- durchlaufe die Menge gegebenenfalls mehrmals, bis bei einem Durchlauf keine Vertauschungen mehr durchgeführt werden mussten

Illustration



55	7	78	12	42
7	55	78	12	42
7	55	78	12	42
7	55	12	78	42
7	55	12	42	78
7	55	12	42	78
7	12	55	42	78
7	12	42	55	78
7	12	42	55	78
7	12	42	55	78
7	12	42	55	78
7	12	42	55	78

sortiert = true

55<7? tausche(7,55); sortiert = false;

55<78?

78<12? tausche(78,12);

78<42? tausche(78,42); Ende: sortiert? sortiert=true;

7<55?

55<12? tausche(55,12); sortiert=false;

55<42? tausche(55,42);

55<78? Ende: sortiert? sortiert=true;

7<12?

12<42?

42<55?

55<78? Ende: sortiert? Fertig.

Implementierung



```
public void sort(int[] array) {
    boolean sortiert;
    do {
        sortiert = true;
        for (int i=1; i<array.length; i++) {
            if (array[i-1]>array[i]) {
                int tmp=array[i-1];
                array[i-1]=array[i];
                array[i]=tmp;
                sortiert=false;
            }
        }
    } while (!sortiert);
}
```

Eigenschaften



- iterativ, nicht rekursiv
- stabil
(gleiche benachbarte Schlüssel werden nicht getauscht)
- in-place
(konstanter zusätzlicher Speicheraufwand)
- effizient für vorsortierte Mengen

Laufzeit



- schlechtester Fall

- Eingabe ist umgekehrt sortiert: $n, n-1, \dots, 2, 1$
- $n-1$ Vertauschungen im ersten Durchlauf (n von Pos. 1 nach n)
- $n-2$ Vertauschungen im zweiten Durchlauf ($n-1$ von Pos. 1 nach $n-1$)
- ...
- 1 Vertauschung im n -ten Durchlauf (2 von Pos. 1 nach 2)

$$\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \in O(n^2)$$

- bester Fall

- Eingabe ist sortiert: $1, 2, \dots, n-1, n$
- ein Durchlauf $O(n)$

- durchschnittlicher Fall $O(n^2)$

Verbesserungen



- Problem
 - große Elemente am Anfang werden schnell nach hinten getauscht, aber kleine Elemente am Ende werden nur langsam nach vorn getauscht
- Lösungsansatz
 - Shaker-Sort (Cocktail-Sort, BiDiBubble-Sort) $O(n^2)$
 - Feld wird abwechselnd von oben und von unten durchlaufen
 - Comb-Sort
 - vergleicht Elemente, die *gap* Indizes auseinanderliegen, $gap > 1$
 - *gap* wird nach jedem Durchlauf reduziert, beispielsweise mit Faktor 0.7, bis $gap = 1$ erreicht
 - für $gap = 1$ entspricht Combsort dann Bubblesort

Überblick



- Einführung
- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Heap-Sort
- Quick-Sort
- Merge-Sort
- Counting-Sort

Prinzip



- durchlaufe die Menge, finde das kleinste Element
- vertausche das kleinste Element mit dem ersten Element
- vorderer Teil ist sortiert (k Elemente nach Durchlauf k), hinterer Teil ist unsortiert (n-k Elemente)
- durchlaufe die hintere, nichtsortierte Teilmenge, finde das n-kleinste Element
- vertausche das n-kleinste Element mit dem n-ten Element

Illustration



a[1]	a[2]	a[3]	a[4]	a[5]
55	7	78	12	42
7	55	78	12	42
7	12	78	55	42
7	12	42	55	78

1. Durchlauf: $7 = \min(1..n)$; tausche 7 mit a[1];
2. Durchlauf: $12 = \min(2..n)$; tausche 12 mit a[2];
3. Durchlauf: $42 = \min(3..n)$; tausche 42 mit a[3];
4. Durchlauf: $55 = \min(4..n)$; tausche 55 mit a[4];

Implementierung



```
private int minimum (int[] array,int anfang,int ende)
{
    int minIdx = anfang;
    for (int index=anfang+1; index<=ende; index++) {
        if (array[index] < array[minIdx]) minIdx = index;
    }
    return minIdx;
}
```

```
public void selectionSort (int[] array) {
    for (int index=0; index<array.length-1; index++) {
        int minIdx = minimum(array,index,array.length-1);
        vertausche(array,index,minIdx);
    }
}
```

Eigenschaften



- iterativ, nicht rekursiv
- instabil
 - gleiche benachbarte Schlüssel werden getauscht
 - lässt sich auch stabil implementieren
- in-place
(konstanter zusätzlicher Speicheraufwand)

Laufzeit



- bester, mittlerer, schlechtester Fall
 - für n Einträge werden n-1 Minima gesucht
 - n-1 Vergleiche für erstes Minimum
 - n-2 Vergleiche für zweites Minimum
 - ...
 - 1 Vergleich für Minimum n-1

$$\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \in O(n^2)$$

Überblick



- Einführung
- Bubble-Sort
- Selection-Sort
- **Insertion-Sort**
- Heap-Sort
- Quick-Sort
- Merge-Sort
- Counting-Sort

Prinzip



- erstes Element ist sortiert, hinterer Teil mit $n - 1$ Elementen ist unsortiert
- entnehme der hinteren, unsortierten Menge ein Element und füge es an die richtige Position der vorderen, sortierten Menge ein ($n-1$ mal)
- Einfügen in die vordere, sortierte Menge erfordert das Verschieben von Elementen

Illustration



5	2	4	6	1	3
---	---	---	---	---	---

Elemente 1..1 sind sortiert, 2..n unsortiert

5	2	4	6	1	3
---	---	---	---	---	---

Vergleiche 2 mit allen Elementen der sortierten Menge beginnend mit dem größten. Wenn ein Element größer als 2 ist, schiebe es eins nach rechts, sonst füge 2 ein.

2	5	4	6	1	3
---	---	---	---	---	---

Elemente 1..2 sind sortiert, 3..n unsortiert

2	5	4	6	1	3
---	---	---	---	---	---

Vergleiche 4 mit allen Elementen der sortierten Menge. Wenn ein Element größer als 4 ist, verschiebe es.

2	4	5	6	1	3
---	---	---	---	---	---

Elemente 1..3 sind sortiert. Einfügen von 6.

2	4	5	6	1	3
---	---	---	---	---	---

Elemente 1..4 sind sortiert. Einfügen von 1 (Dazu werden Elemente 6,5,4 und 2 jeweils um eins nach rechts verschoben. Danach wird 1 an Pos. eins eingefügt.

1	2	4	5	6	3
---	---	---	---	---	---

...

1	2	3	4	5	6
---	---	---	---	---	---

Implementierung



```
public int insertionSort (int[] array) {  
    for (int j=1 to array.length-1) {  
        int key = array[j];  
        int i = j-1;  
        while (i>=0 && array[i]>key) {  
            array[i+1] = array[i];  
            i = i-1;  
        }  
        array[i+1] = key;  
    }  
}
```

Vorsicht mit i=-1

Eigenschaften



- iterativ, nicht rekursiv
- stabil
 - gleiche benachbarte Schlüssel werden nicht getauscht
- in-place
(konstanter zusätzlicher Speicheraufwand)
- effizient für vorsortierte Mengen

Laufzeit



- **bester Fall**
 - Menge ist vorsortiert
 - innere while-Schleife wird nicht durchlaufen
 - $O(n)$
- **schlechtester Fall**
 - Menge ist umgekehrt sortiert
 - k-1 Verschiebeoperationen für das k-te Element
 - $O(n^2)$
- **mittlerer Fall**
 - $O(n^2)$

Verbesserungen



- Problem
 - Elemente müssen teilweise über große Bereiche verschoben werden
- Lösungsansatz
 - Shell-Sort
 - führe Insertionsort für z. B. jedes 4. Element, dann für jedes 2. Element und dann für jedes Element durch
 - statt 1,2,4, ..., 2^k wird auch 1,4, 13, ..., $3(k-1)+1$ verwendet
 - Sortieren von Feld $a[0..2k-1]$:
 - sortiere $a[0], a[4], a[8], a[12], \dots$
 - sortiere $a[0], a[2], a[4], a[6], \dots$
 - sortiere $a[0], a[1], a[2], a[3], \dots$
 - reduzierte Zahl von Verschiebeoperationen $O(n \log (n^2))$

Überblick



- Einführung
- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- **Heap-Sort**
- Quick-Sort
- Merge-Sort
- Counting-Sort

Prinzip



- Repräsentation der Daten durch einen Max-Heap
 - Binärbaum, in dem der Schlüssel eines Knoten immer größer ist als die Schlüssel der beiden Nachfolgeknoten
 - Wurzelknoten enthält größtes Element
- Iteratives Vorgehen
 - Entnahme des Wurzelknotens als größtes Element
 - Herstellen eines Max-Heaps aus den verbleibenden Elementen
- Motivation
 - n Iterationen mit Aufwand $\log(n)$
 - n Elemente müssen entnommen werden
 - **Wiederherstellung des Heaps in $\log(n)$**

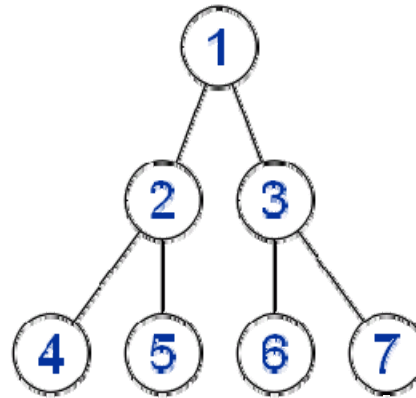
Illustration - Heap



- Felder eines Feldes können als Binärbaum angeordnet werden



Indizes eines Feldes



Repräsentation der Indizes durch einen Binärbaum

- Für Indizes im Baum gilt
 - Indizes eines Nachfolgeknotens k ergeben sich als $2k$ und $2k+1$
- Baum wird zum besseren Verständnis betrachtet
- Implementierung erfolgt in-place auf dem Feld

Illustration - Heap



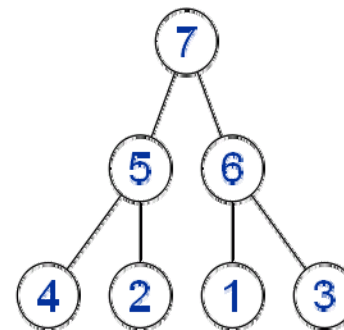
äquivalente
Aussagen

- Folge von Daten ist ein Max-Heap, wenn
 - die Schlüssel beider Nachfolgeknoten eines jeden Knotens k kleiner sind als der Schlüssel von Knoten k
 - der Schlüssel des Feldes an Stelle k kleiner ist als die Schlüssel an den Stellen $2k$ und $2k+1$
 - über die Relationen anderer Knoten ist nichts bekannt
 - aus den Anforderungen ergibt sich, dass der Wurzelknoten den größten Schlüssel enthält

Beispiel: Folge ist ein Max-Heap

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

Feldrepräsentation



Baumrepräsentation

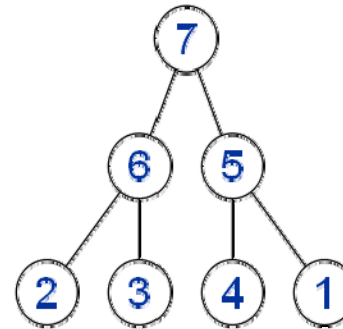
Beispiele - Heap



Beispiel: gleiche Menge, alternative Anordnung, Folge ist ein Max-Heap

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

Feldrepräsentation

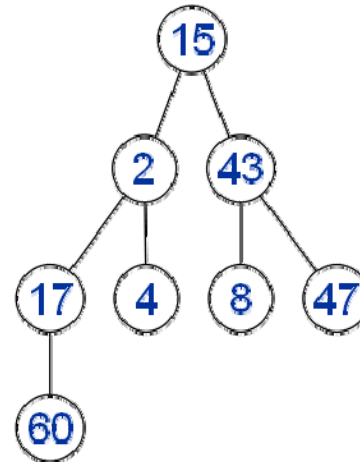


Baumrepräsentation

Beispiel: Folge ist kein Max-Heap

15	2	43	17	4	8	47	60
----	---	----	----	---	---	----	----

Feldrepräsentation



Baumrepräsentation

- Motivation für Heap-Sort:
 - Initialisierung eines Max-Heap in $O(n)$
 - Wiederherstellung nach Löschen der Wurzel in $O(\log n)$

Prinzip - Wiederholung

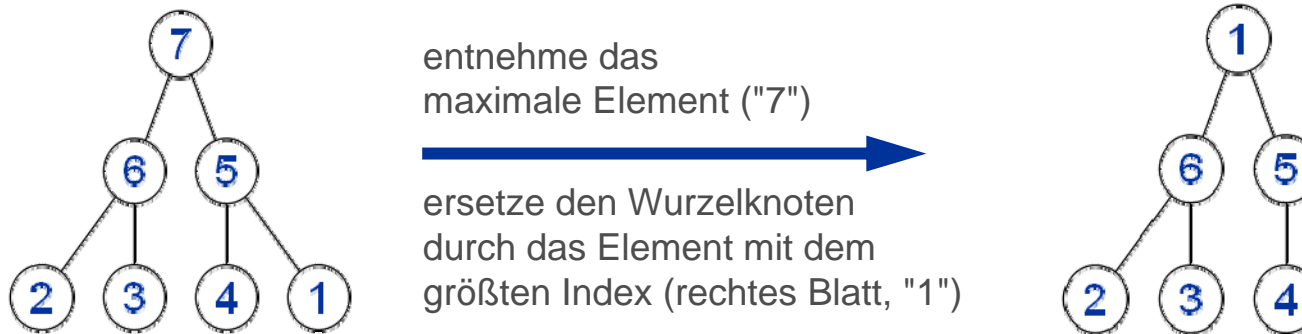


- **Initialisierung**
 - Aufbau eines Max-Heaps für ein Feld in $O(n)$
- **n Iterationen**
 - Entnahme des größten Elements (Wurzelknoten)
 - Wiederherstellung des Max-Heaps aus den verbleibenden Elementen in $O(\log n)$

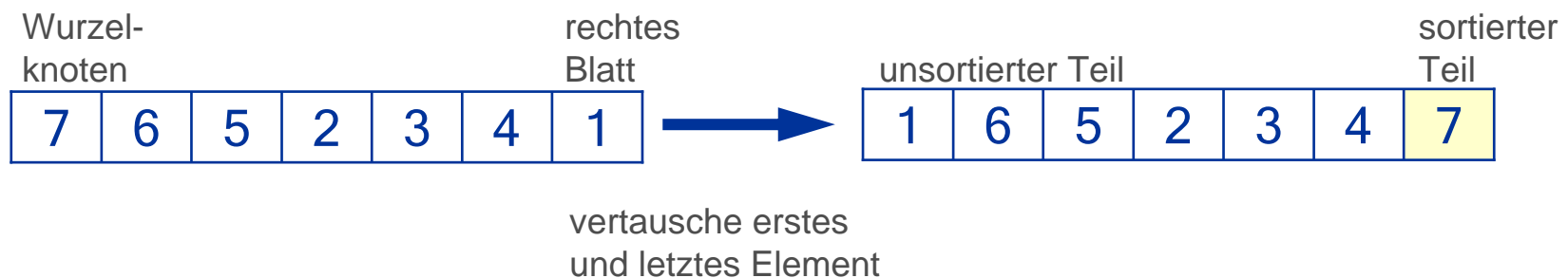
Schritt



■ Heap-Repräsentation



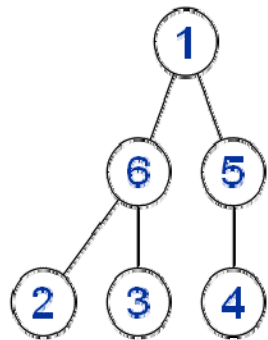
■ Feldrepräsentation



Schritt



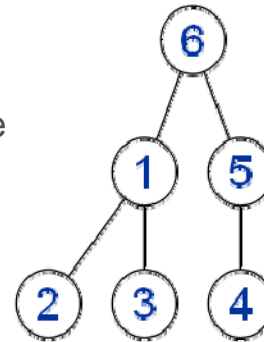
- Wiederherstellung eines Max-Heaps (versickern eines Elements, percolate)
- Heap-Repräsentation



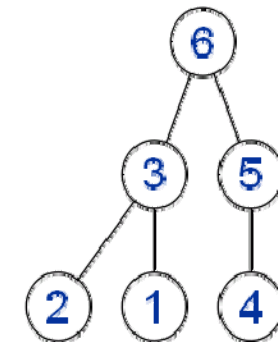
- Versickern des Wurzelknotens

- vergleiche das zu versickernde Element mit dem Maximum der beiden nachfolgenden Knoten

- tausche Wurzelknoten mit dem Maximum aller drei Elemente



- wiederhole Vergleiche für das zu versickernde Element



wiederhergestellter Max-Heap (Maximum steht in der Wurzel)

- maximal $\log n$ Versickerungs-Schritte

Schritt



- Wiederherstellung eines Max-Heaps (versickern eines Elements, percolate)
- Feldrepräsentation

	1	2	3	4	5	6	7
a	1	6	5	2	3	4	7

- Versickern des ersten Elements $a[1]$
- Vergleiche mit Maximum der Elemente $a[2 \cdot 1]$ und $a[2 \cdot 1 + 1]$
- Tausche entsprechend

	1	2	3	4	5	6	7
a	6	1	5	2	3	4	7

- Vergleiche $a[2]$ mit Maximum der Elemente $a[2 \cdot 2]$ und $a[2 \cdot 2 + 1]$
- Tausche entsprechend

	1	2	3	4	5	6	7
a	6	3	5	2	1	4	7

wiederhergestellter
Max-Heap (Maximum
von $a[1..6]$ steht in $a[1]$)
 $a[7]$ ist bereits sortiert

- maximal $\log n$ Versickerungs-Schritte

Versickern



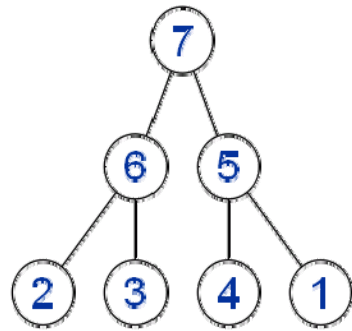
- vertausche mit dem größeren der Nachfolger, solange dieser größer als das zu versickernde Element ist
- Versickern des ersten Elements eines Feldes a in den Grenzen j und t

```
void percolate (int[] a, int j, int t) {  
    while (int h=2*j<=t) {  
        if (h<t && a[h+1]>a[h]) h++;  
        if (a[h]>a[j]) {  
            swap(a,h,j);  
            j=h;  
        }  
        else break;  
    }  
}
```

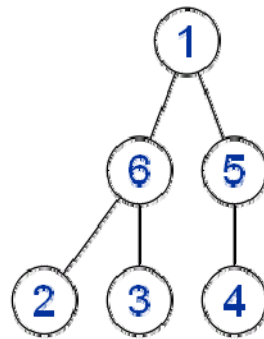
Nachfolgerindex im gültigen Bereich?
h zeigt auf größeres der Nachfolgerelemente
tausche ggf. erstes Element a[j] mit a[h]
aktualisiere Index j des zu versickernden El.

while-Schleife wird maximal log-n-mal durchlaufen.

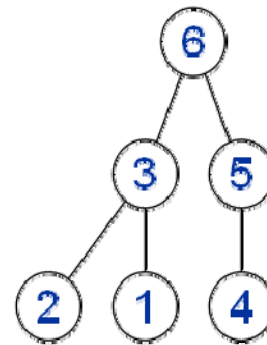
Schritte - Illustration



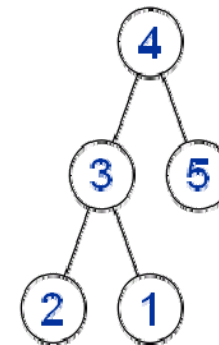
entnehme "7"



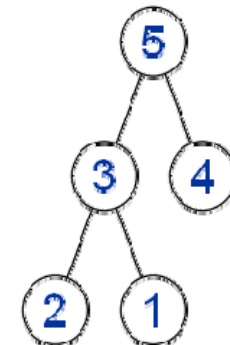
"1" neue Wurzel



versickere "1"



entnehme "6",
"4" neue Wurzel



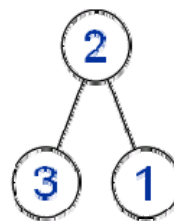
versickere "4"



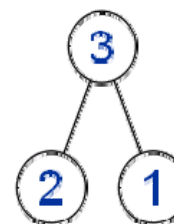
entnehme "5",
"1" neue Wurzel



versickere "1"



entnehme "4",
"2" neue Wurzel



versickere "2"



Schritte - Illustration



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

entnehme "7", setze "1" an erste Position

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

versickere "1"

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

entnehme "6", setze "4" an erste Position

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

versickere "4"

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

entnehme "5", setze "1" an erste Position

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

versickere "1"

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

entnehme "4", setze "2" an erste Position

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

versickere "2"

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

...

Implementierung



```
■ void heapSort (int[] a) {
    int j, hi = a.length-1;

    for (j=hi/2; j>=1; j--)
        percolate(a, j, hi);

    for (j=hi; j>1; j--) {
        swap(a, 1, j);
        percolate(a, 1, j-1);
    }
}
```

Initialisierung:

Versickere die ersten $n/2$ Elemente in umgekehrter Reihenfolge. Die zweite Hälfte genügt automatisch einem Max-Heap.

$O(n)$ Iterationen

$O(1)$ entnehme das Maximum

$O(\log n)$ versickere Element

Aufwand der Schleife:

$O(n \log n)$

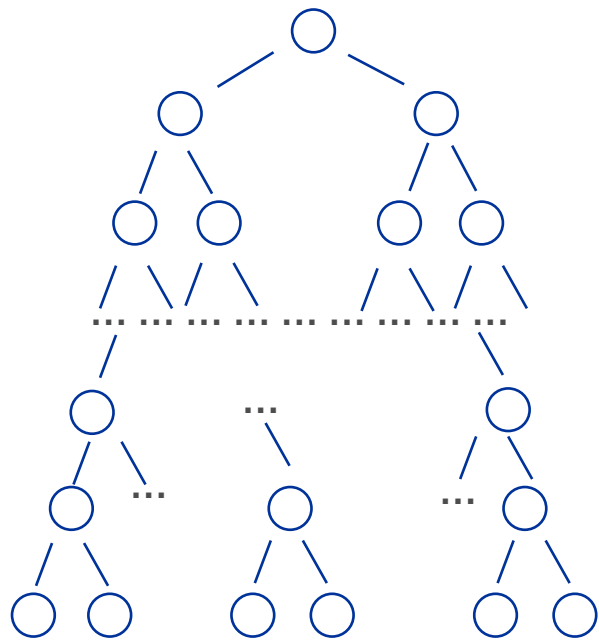
Aufwand der Initialisierung:

Versickere $n/2$ Elemente

Konservative Abschätzung $O(n \log n)$

⇒ Gesamtaufwand $O(n \log n)$

Aufwand der Initialisierung



Ebene	Knoten	Aufwand (zwei Vergleich pro Knoten und Ebene)
k	2^0	$2 \cdot k$
k-1	2^1	$2 \cdot (k-1)$
k-2	2^2	$2 \cdot (k-2)$
...
2	2^{k-2}	$2 \cdot 2$
1	2^{k-1}	$2 \cdot 1$
0	2^k	$2 \cdot 0$

Zahl der Elemente: $n = \sum_{i=0}^k 2^i = 2^{k+1} - 1$

Gesamtkosten: $2 \cdot (2^0 \cdot k + 2^1 \cdot (k-1) + \dots + 2^{k-1} \cdot 1 + 2^k \cdot 0)$

Aufwand der Initialisierung



$$\sum_{i=1}^k (i \cdot 2^{k-i}) = 2 \cdot \sum_{i=1}^k (i \cdot 2^{k-i}) - \sum_{i=1}^k (i \cdot 2^{k-i})$$

$$a = 2a - a$$

$$= 1 \cdot 2^k + 2 \cdot 2^{k-1} + 3 \cdot 2^{k-2} + \dots + (k-1) \cdot 2^2 + k \cdot 2^1 \\ - (1 \cdot 2^{k-1} + 2 \cdot 2^{k-2} + 3 \cdot 2^{k-3} + \dots + (k-1) \cdot 2^1 + k \cdot 2^0)$$

$$= 2^k + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2^1 - k$$

$$= 2^k + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2^1 + 2^0 - 2^0 - k$$

$$= 2^{k+1} - 1 - 2^0 - k$$

$$= n - 2^0 - k$$

$$= n - 2^0 - (\log(n+1) - 1) \in O(n)$$

geometrische
Reihe

$$n = 2^{k+1} - 1$$

$$k = \log(n+1) - 1$$

Gesamtaufwand für
das Versickern **aller**
Knoten

Laufzeit



```
■ void heapSort (int[] a) {  
    int j, hi = a.length-1;  
    for (j=hi/2; j>=1; j--)  
        percolate(a, j, hi);  
    for (j=hi; j>1; j--)  
    {  
        swap(a, 1, j);  
        percolate(a, 1, j-1);  
    }  
}
```

Initialisierung:
 $O(n)$

Aufwand der Schleife:
 $O(n \log n)$

- bester, mittlerer, schlechtester Fall
 $O(n \log n)$

Eigenschaften



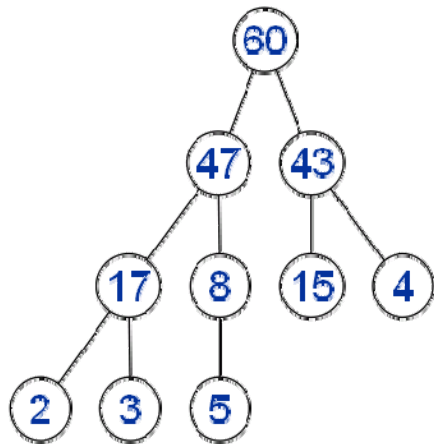
- iterativ, nicht rekursiv
- nicht stabil
 - gleiche benachbarte Schlüssel werden eventuell getauscht
- in-place
(konstanter zusätzlicher Speicheraufwand)
- optimale Effizienz von $O(n \log n)$

Verbesserungen

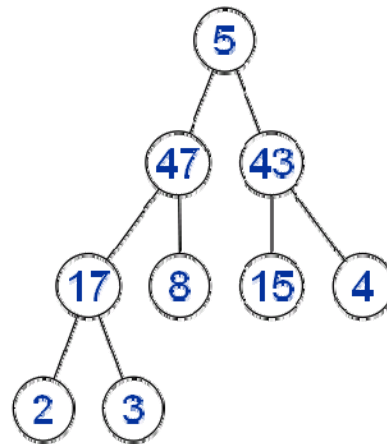


- Problem
 - Elemente müssen über viele Ebenen versickert werden (2 Vergleichsoperationen pro Element und Ebene)
- Lösungsansatz
 - verringere die Zahl der Vergleichsoperationen beim Versickern
 - Bottom-up Heap Sort
 - vergleiche nur die beiden Nachfolgeelemente (1 Vergleich)
 - tausche das zu versickernde Element in jedem Fall mit dem maximalen Nachfolgeelement (0 Vergleiche)
 - bewege Element eventuell wieder etwas nach oben, wenn Blatt erreicht
 - Verbesserung der Effizienz um konstanten Faktor

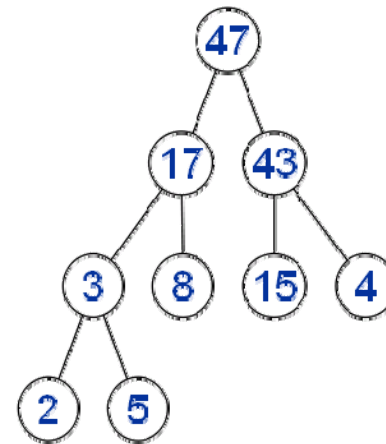
Beispiel



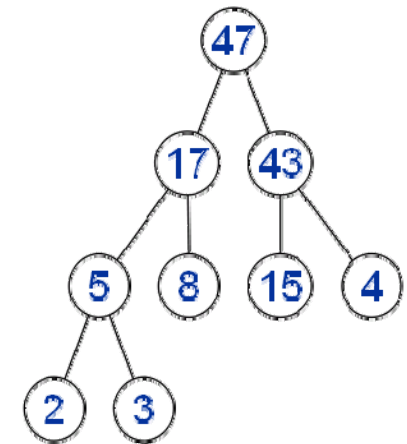
entnehme "60",
"5" neue Wurzel



tausche "5" in jedem Fall
mit dem maximalen Nach-
folger (1 statt 2 Vergleiche)



bewege "5" nach oben,
wenn Vorgänger kleiner



■ Bottom-up Heap Sort

- es ist günstiger, Elemente mit nur einem statt zwei Vergleichen bis zu einem Blatt zu versickern und dann eventuell etwas nach oben zu korrigieren

Bottom-up Heap Sort



```
■ void percolateB (int[] a, int j, int t) {  
    while (int h=2*j<=t) {  
        if (h<t && a[h+1]>a[h]) h++;  
        if (a[h]>a[j]) {  
            swap(a,h,j);  
            j=h;  
        }  
        else break;  
    }  
    bubbleUp (a, j);  
}
```

Bottom-up Heap Sort



- ```
void bubbleUp (int[] a, int j) {
 int x = a[j];
 for (; j>1 && a[j/2]<x; j/=2)
 a[j] = a[j/2];
 a[j] = x;
}
```
- Ziehe kleinere Elemente  
jeweils eine Ebene nach unten.
- Plaziere x an die  
korrekte Position.

Ist bestimmt effizient, aber auch ein schönes Beispiel dafür,  
dass die Lesbarkeit bei zuviel Effizienz leiden kann.

# Überblick



- Einführung
- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Heap-Sort
- Quick-Sort
- Merge-Sort
- Counting-Sort

# Prinzip



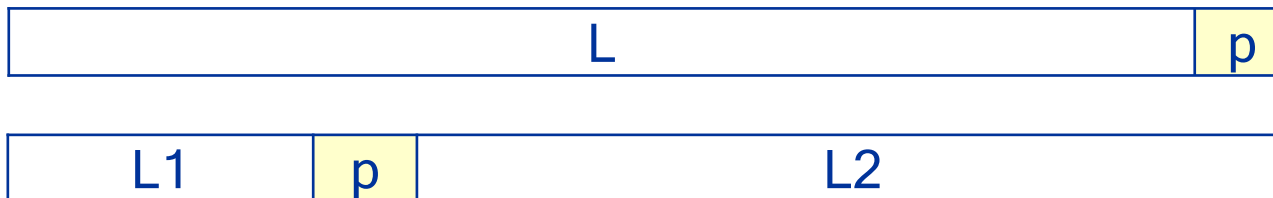
- Teile-und-Herrsche-Ansatz
  - aufwendiger Teile-Schritt, einfacher Merge-Schritt
- Mengen mit einem oder keinem Element sind sortiert
- ansonsten
  - Aufteilung des Problems in zwei Teilmengen, wobei alle Elemente einer Teilmenge kleiner als alle Elemente der anderen Teilmenge sind
  - rekursiver Aufruf des Algorithmus für beide Teilmengen
- Verbindung der Teilergebnisse
  - implizit gegeben, da Sortierung der Teilmengen in-place
- Aufteilung in Teilmengen
  - Wahl eines Schlüssels / Pivotelements
  - Elemente, die kleiner als das Pivotelement sind, werden der linken Teilmenge zugeordnet
  - Elemente, die größer als das Pivotelement sind, werden der rechten Teilmenge zugeordnet

# Prinzip - Illustration



- Quicksort (L)  
if ( |L| <= 1 ) return L;  
else  
    wähle Pivotelement p aus L;  
    L1 = { a in L | a < p };  
    L2 = { a in L | a > p };  
    Quicksort (L1);  
    Quicksort (L2);

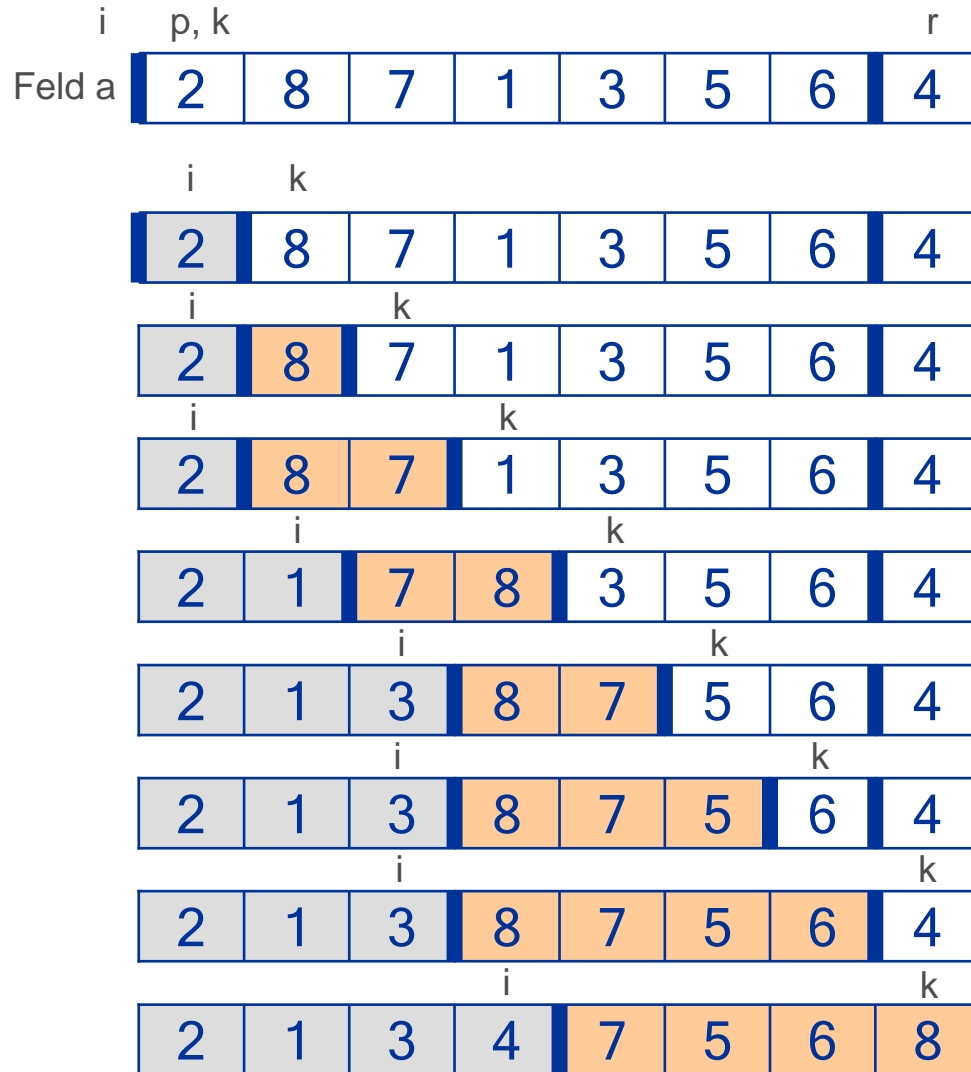
Aufteilung in zwei  
Teilmengen L1, L2



kein merge-Schritt, da  
in-place-Sortierung:  
L1 < p < L2

# Aufteilung in Teilmengen

## Illustration



p – erstes Element,  
r – letztes Element  
(Pivotelement)

i – letztes Element von L1  
k – erstes Element hinter L2

$a[k] \leq a[r] \rightarrow i+=1; \text{tausche}(a[i], a[k]); k+=1;$

$a[k] > a[r] \rightarrow k+=1;$

$a[k] > a[r] \rightarrow k+=1;$

$a[k] \leq a[r] \rightarrow i+=1; \text{tausche}(a[i], a[k]); k+=1;$

$a[k] \leq a[r] \rightarrow i+=1; \text{tausche}(a[i], a[k]); k+=1;$

$a[k] > a[r] \rightarrow k+=1;$

$a[k] > a[r] \rightarrow k+=1;$

$i+=1; \text{tausche}(a[i], a[k]);$

# Implementierung



```
int aufteilung (int[] a, int p, int r) {
 int pivot = a[r];
 int i = p - 1;
 for (k = p; k <= r-1; k++)
 if (a[k]<=pivot) { i+=1; tausche(a[i],a[k]); }
 i+=1; tausche (a[i],a[r]);
 return i;
}
```

```
void quickSort (int[] a, int p, int r) {
 if (p<r) {
 int i = aufteilung (a, p, r);
 quickSort (a, p, i-1);
 quickSort (a, i+1, r);
 }
}
```

# Laufzeit



```
int aufteilung (int[] a, int p, int r) {
 int pivot = a[r];
 int i = p - 1;
 for (k = p; k <= r-1; k++)
 if (a[k]<=pivot) { i+=1; tausche(a[i],a[k]); }
 i+=1; tausche (a[i],a[k]);
 return i;
}
```

O(n)

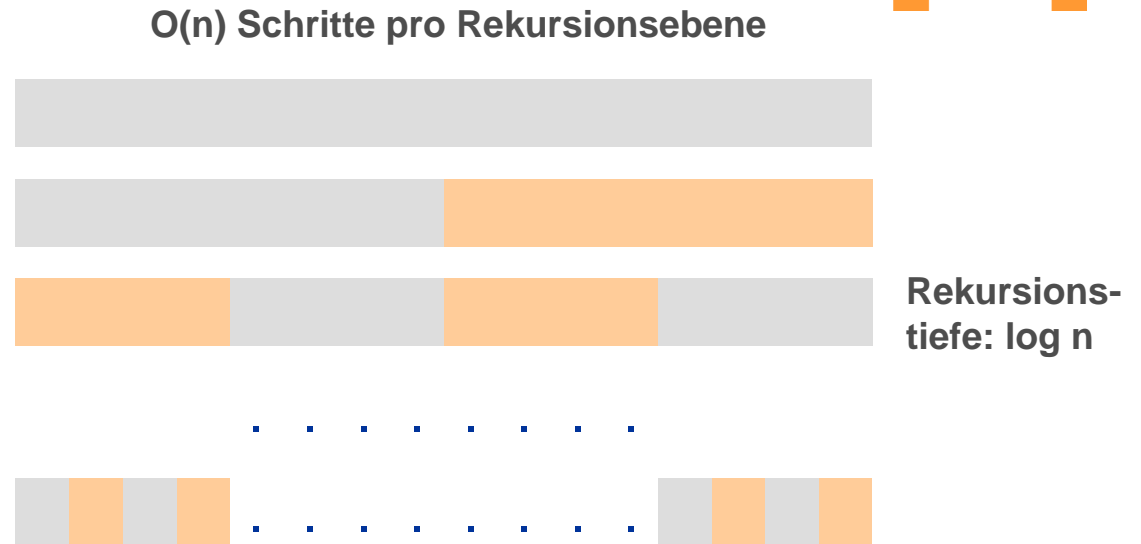
```
void quickSort (int[] a, int p, int r) {
 if (p<r) {
 int i = aufteilung (a, p, r);
 quickSort (a, p, i-1);
 quickSort (a, i+1, r);
 }
}
```

O(n)  
T(n-k)  
T(k)

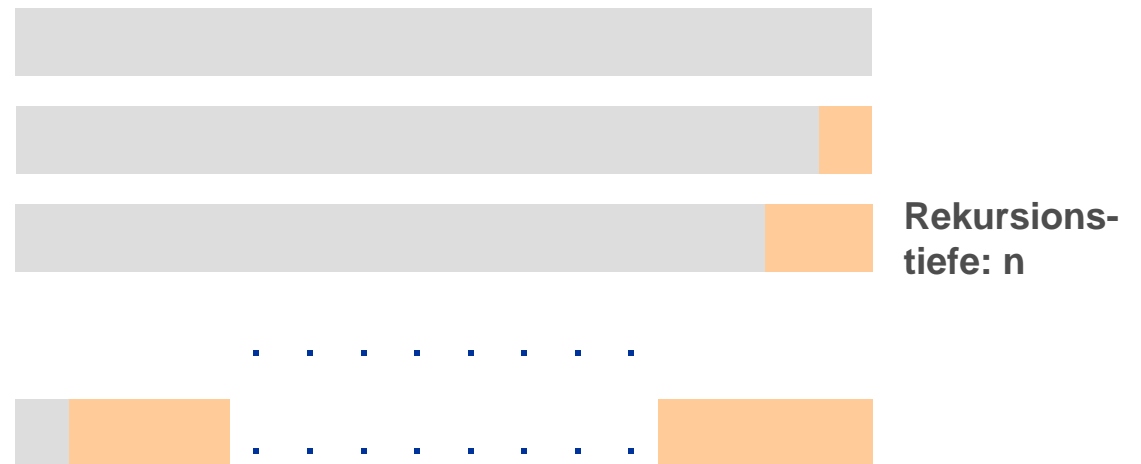
# Laufzeit



- **bester Fall:**  
Aufteilung  
 $n \rightarrow (n/2) + (n/2)$   
 $O(n \log n)$



- **schlechtester Fall:**  
Aufteilung  
 $n \rightarrow (n-1) + (1)$   
 $O(n^2)$



# Laufzeit



- bester Fall: Master-Theorem für  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$T(n) \in \Theta\left(n^{\log_b a} \log n\right) \text{ falls } f(n) \in \Theta\left(n^{\log_b a}\right)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$a = 2, \quad b = 2, \quad f(n) = O(n), \quad \log_b a = \log_2 2 = 1$$

$$f(n) \in \Theta\left(n^{\log_2 2}\right) \rightarrow T(n) \in \Theta(n \log n)$$

- mittlerer Fall:  $O(n \log n)$

# *Eigenschaften*



- rekursiv
- in-place  
(konstanter zusätzlicher Speicheraufwand)
- nicht stabil
- optimale mittlere Laufzeit von  $O(n \log n)$
- ungünstige schlechteste Laufzeit von  $O(n^2)$

# Verbesserungen



- Vermeidung eines schlechten Pivotelements
  - Suchen nach dem mittleren Schlüsselwert nicht möglich, da nicht in konstanter Zeit
- Alternativ
  - mittleres Element von (erster Schlüssel, mittlerer Schlüssel, letzter Schlüssel)
  - randomisierte Wahl des Pivotelements (große Wahrscheinlichkeit zur Vermeidung des schlechtesten Falls)

# Überblick



- Einführung
- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Heap-Sort
- Quick-Sort
- Merge-Sort
- Counting-Sort

# Prinzip



- Teile-und-Herrsche-Ansatz
- Mengen mit einem oder keinem Element sind sortiert
- ansonsten
  - Aufteilung des Problems in zwei gleich große Teilmengen
  - rekursiver Aufruf des Algorithmus für beide Teilmengen
  - Verbindung der Teilmengen
- Motivation gegenüber Quick-Sort
  - simple Unterteilung im Vergleich zu Quick-Sort
  - optimale Rekursionstiefe von  $\log n$
  - aufwendiger merge-Schritt



# Implementierung



```
void mergeSort (int[] a, int l, int h) {
 if (l < h) {
 int m = (l+h) / 2;
 mergeSort (a, l, m-1);
 mergeSort (a, m, h);
 int[] tmp = new int[h-l+1];
 for (int i=0, j=l, k=m; i<tmp.length;)
 if ((k>h) || ((j<m) && (a[j]<a[k])))
 tmp[i++] = a[j++];
 else
 tmp[i++] = a[k++];
 for (int i=0; i<tmp.length; i++)
 a[l+i]=tmp[i];
 } // else ... Trivialfall, tue nichts
}
```

Verbindung  
der Teillösungen

- vergleiche die kleinsten  
Schlüssel der Teillösungen  
- kleinster Schlüssel wird  
nach tmp kopiert

kopiere tmp zurück nach a

# Laufzeit



```
void mergeSort (int[] a, int l, int h) {
 if (l < h) {
 int m = (l+h) / 2;
 mergeSort (a, l, m-1); T (n/2)
 mergeSort (a, m, h); T (n/2)
 int[] tmp = new int[h-l+1];
 for (int i=0, j=l, k=m; i<tmp.length; i) O(n)
 if ((k>h) || (j<m) && (a[j]<a[k]))
 tmp[i++] = a[j++];
 else
 tmp[i++] = a[k++];
 for (int i=0; i<tmp.length; i++)
 a[l+i]=tmp[i];
 }
}
```

# Laufzeit



- bester, mittlerer, schlechtester Fall

Master-Theorem für  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$T(n) \in \Theta\left(n^{\log_b a} \log n\right)$  falls  $f(n) \in \Theta\left(n^{\log_b a}\right)$

$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

$a = 2, \quad b = 2, \quad f(n) = O(n), \quad \log_b a = \log_2 2 = 1$

$f(n) \in \Theta\left(n^{\log_2 2}\right) \rightarrow T(n) \in \Theta(n \log n)$

# *Eigenschaften*



- rekursiv
- je nach Implementierung in-place oder zusätzlicher Speicheraufwand von  $O(n)$  oder  $O(n \log n)$
- stabil
- optimale mittlere Laufzeit von  $O(n \log n)$
- beste und schlechteste Laufzeit von  $O(n \log n)$
- im Gegensatz zu Quick-Sort immer garantierte Aufteilung in zwei gleichgroße Teilprobleme

# Verbesserungen



- Generieren und Löschen von tmp ist relativ teuer
  - einmaliges Generieren
  - Vermeiden des Rück-Kopierens
- Verbinden der Teillösungen
  - wenn eine Teillösung vollständig abgearbeitet ist, kann die andere Teillösung ohne Vergleiche kopiert werden
- trotz des besseren schlechtesten Falls gegenüber Quick-Sort wird Quick-Sort in der Praxis bevorzugt

# Überblick



- Einführung
- Bubble-Sort
- Selection-Sort
- Insertion-Sort
- Heap-Sort
- Quick-Sort
- Merge-Sort
- Counting-Sort

# Prinzip



- vereinfachte Sortierung bei bekanntem Wertebereich  
 $a[i] = 0 \dots k$
- wenn in einem Feld  $a[0..n] = 0..n$  jeder Schlüssel genau einmal vorkommt, kann jeder Schlüssel **ohne Vergleichsoperationen** an die korrekte Stelle plaziert werden  $s[a[i]] = a[i]$
- Counting-Sort basiert auf dem Prinzip der geschickten Plazierung von Elemente ohne Vergleichsoperationen, kann aber auch mehrfaches Auftreten einzelner Schlüssel behandeln

# Implementierung



```
void countingSort (int[] a, int[] b, int max)
{
 int i; int[] c = new int[max+1];

 for (i=0; i<=max; i++) c[i]=0;
 for (i=0; i<a.length; i++) c[a[i]]++;

 for (i=1; i<=max; i++)
 c[i]=c[i]+c[i-1];

 for (i=a.length; i>=1; i--) {
 b[c[a[i]]-1]=a[i];
 c[a[i]]--;
 }
}
```

c [ i ] gibt an,  
wie oft Schlüssel i auftritt.

c [ i ] gibt an,  
wieviel Elemente  
einen Schlüssel kleiner  
oder gleich i haben.

Plaziere jeden Schlüssel an  
die korrekte Position im Feld b.  
Aktualisiere c entsprechend

# Illustration



a

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 2 | 0 | 3 | 3 | 0 | 3 |

c

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 3 | 0 | 1 |

Erste Initialisierung  
von c.  $c[i]$  gibt an, wie  
oft Schlüssel i in a  
enthalten ist.

c

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 2 | 4 | 7 | 7 | 8 |

Zweite Initialisierung  
von c.  $c[i]$  gibt an, wieviel  
Elemente von a einen  
Schlüssel enthalten, der  
kleiner oder gleich i ist.

Daraus können Schlussfolgerungen für die Positionen von Schlüsseln im sortierten Feld abgeleitet werden. z. B. kann ein Schlüssel "5" an die 8-te Position platziert werden, da 8 Elemente kleiner oder gleich "5" sind. "0" kann an Position 2 platziert werden usw.

Nach der Platzierung eines Schlüssels i wird  $c[i]$  um eins reduziert, um das mehrfache Auftreten eines Schlüssels zu behandeln.

z. B. "0" wird an Position 2 platziert.  $c[0]$  wird um eins reduziert, d. h. die nächste "0" wird an Position 1 platziert.

# Illustration



|   |                                    |   |                            |   |                     |
|---|------------------------------------|---|----------------------------|---|---------------------|
| a | 0 1 2 3 4 5 6 7<br>2 5 2 0 3 3 0 3 | c | 0 1 2 3 4 5<br>2 2 4 7 7 8 | b | 0 1 2 3 4 5 6 7<br> |
|---|------------------------------------|---|----------------------------|---|---------------------|

$a[0] = 2; c[a[0]]-1=3 \rightarrow b[3]=2; c[2]--$

|   |                                    |   |                            |   |                      |
|---|------------------------------------|---|----------------------------|---|----------------------|
| a | 0 1 2 3 4 5 6 7<br>2 5 2 0 3 3 0 3 | c | 0 1 2 3 4 5<br>2 2 3 7 7 8 | b | 0 1 2 3 4 5 6 7<br>2 |
|---|------------------------------------|---|----------------------------|---|----------------------|

$a[1] = 5; c[a[1]]-1=7 \rightarrow b[7]=5; c[5]--$

|   |                                    |   |                            |   |                         |
|---|------------------------------------|---|----------------------------|---|-------------------------|
| a | 0 1 2 3 4 5 6 7<br>2 5 2 0 3 3 0 3 | c | 0 1 2 3 4 5<br>2 2 3 7 7 7 | b | 0 1 2 3 4 5 6 7<br>2  5 |
|---|------------------------------------|---|----------------------------|---|-------------------------|

$a[2] = 2; c[a[2]]-1=2 \rightarrow b[2]=2; c[2]--$

|   |                                    |   |                            |   |                           |
|---|------------------------------------|---|----------------------------|---|---------------------------|
| a | 0 1 2 3 4 5 6 7<br>2 5 2 0 3 3 0 3 | c | 0 1 2 3 4 5<br>2 2 2 7 7 7 | b | 0 1 2 3 4 5 6 7<br>2 2  5 |
|---|------------------------------------|---|----------------------------|---|---------------------------|

$a[3] = 0; c[a[3]]-1=1 \rightarrow b[1]=0; c[0]--$

.....

# Laufzeit



```
void countingSort (int[] a, int[] b, int max)
{
 int i; int[] c = new int[max+1];

 for (i=0; i<=max; i++) c[i]=0; O (max)
 for (i=0; i<a.length; i++) c[a[i]]++; O (n)

 for (i=1; i<=max; i++) O (max)
 c[i]=c[i]+c[i-1];

 for (i=a.length-1; i>=0; i--) { O (n)
 b[c[a[i]]-1]=a[i];
 c[a[i]]--;
 }
}
```

**Gesamtlaufzeit  $O(\max + n)$  bzw.  $O(n)$ , wenn  $\max = O(n)$**

# *Eigenschaften*



- iterativ
- zusätzlicher Speicheraufwand von  $O(n)$
- stabil
  - Zu sortierendes Feld  $a$  wird von hinten nach vorn bearbeitet. Andernfalls wäre das Verfahren nicht stabil, da gleiche Elemente voreinander in Feld  $b$  abgelegt werden.
- Laufzeit von  $O(n + k)$ , wobei  $k$  die Größe des Wertebereichs der Elemente des zu sortierenden Feldes angibt
- verwendet keinerlei Vergleichsoperationen
  - daher gilt die untere Schranke von  $O(n \log n)$  nicht zwangsläufig

# Erweiterung – Radix-Sort



- Beispiel:
  - Postleitzahl kann als Menge von fünf Schlüsseln mit Wertebereich 0..9 betrachtet werden, Geburtstag besteht aus drei Schlüsseln
  - Sortierung kann durch eine **Sequenz stabiler Sortierverfahren** realisiert werden:
    - Counting-Sort der letzten Ziffer
    - Counting-Sort der vorletzten Ziffer,
    - ...,
    - Counting-Sort der ersten Ziffer
- `radixSort (A, d)`
  - `for i=1 to d do countingSort(A, i);` sortiere nach Schlüssel i

# Variante – Bucket-Sort



- sehr ähnlich zu Counting-Sort
- wird verwendet, wenn wenig verschiedene Schlüsselwerte häufig auftreten
  - $k$  verschiedene Schlüsselwerte,
  - $n$  Elemente
  - $k \ll n$
- Prinzip
  - ermittle die Häufigkeit jedes Schlüssels
  - daraus ergibt sich die Position im sortierten Feld

# Implementierung



Schlüsselwerte 0..anzahlBuckets

```
void bucketsort(int z[], int anzahlBuckets) {

 int buckets[] = new int[anzahlBuckets];
 for (int i=0; i<z.length; i++) {
 buckets[z[i]]++;
 }

 int x=0;
 for (int i=0; i<anzahlBuckets; i++) {
 while (buckets[i] > 0) {
 z[x++] = i;
 buckets[i]--;
 }
 }
}
```

erstelle  
Histogramm:  
buckets [ i ] gibt an,  
wie oft Schlüssel i  
auftritt.

Schlüssel "0" wird an den  
ersten buckets [ 0 ] Positionen  
in z eingetragen.

Schlüssel "1" wird an den  
folgenden buckets [ 1 ]  
Positionen in z eingetragen.

Schlüssel "2" wird an den  
folgenden buckets [ 2 ]  
Positionen in z eingetragen.

Einschränkung: Funktioniert nicht für Datensätze mit Satellitendaten.

...

# Zusammenfassung

## Vergleichsbasiertes Sortieren



- beweisbare untere Schranke von  $\Omega ( n \log n )$
- Bubble-, Selection-, Insertion-Sort
  - intuitive Strategien (Vertauschen benachbarter Elemente, Auswahl des Minimums und Anfügen an sortierte Teilmenge, Einsortieren eines Elements in sortierte Teilmenge)
- Heap-Sort
  - basiert auf einer speziellen Datenstruktur mit schnellem Zugriff auf das maximale Element einer Menge (Max-Heap)
  - n-malige Entnahme des maximalen Elements, wobei der restliche Max-Heap jedes Mal in  $O ( \log n )$  aktualisiert werden kann.

# Zusammenfassung

## Vergleichsbasiertes Sortieren



- Quick-, Merge-Sort
  - Teile-und-Herrsche-Ansätze
  - bei Quick-Sort ist das Aufteilen aufwendig in  $O(n)$ , aber das Zusammenfügen der Teillösungen implizit gegeben
  - bei Merge-Sort ist das Aufteilen einfach, aber das Zusammenfügen aufwendig in  $O(n)$
  - Quick-Sort hat einen schlechteren schlechtesten Fall als Merge-Sort

# Zusammenfassung

## Nicht-vergleichsbasiertes Sortieren



- Counting-, Radix-, Bucket-Sort
  - Schlüsselwerte müssen in eingeschränktem Wertebereich  $0..k$  liegen
  - keine Vergleichsoperationen
  - Position eines Elements wird mit Hilfe eines Histogramms ermittelt
  - effizient, Laufzeit in  $O(n)$

# Zusammenfassung



| Verfahren | bester Fall $O(x)$ | mittlerer Fall $O(x)$ | schlechtester Fall $O(x)$ | stabil | rekursiv | Speicher $O(x)$ |
|-----------|--------------------|-----------------------|---------------------------|--------|----------|-----------------|
| Bubble    | $n$                | $n^2$                 | $n^2$                     | ja     | nein     | 1               |
| Selection | $n^2$              | $n^2$                 | $n^2$                     | ja     | nein     | 1               |
| Insertion | $n$                | $n^2$                 | $n^2$                     | ja     | nein     | 1               |
| Heap      | $n \log n$         | $n \log n$            | $n \log n$                | nein   | nein     | 1               |
| Quick     | $n \log n$         | $n \log n$            | $n^2$                     | nein   | ja       | 1               |
| Merge     | $n \log n$         | $n \log n$            | $n \log n$                | ja     | ja       | $n$             |
| Counting  | $n$                | $n$                   | $n$                       | ja     | nein     | $n$             |
| Radix     | $n$                | $n$                   | $n$                       | ja     | nein     | $n$             |
| Bucket    | $n$                | $n$                   | $n$                       | ja     | nein     | $n$             |

# *Nächstes Thema*



- Algorithmen
  - Suchverfahren

# *Zusätzliche Referenzen*



- <http://de.wikipedia.org/wiki/Sortierverfahren>
- <http://de.wikipedia.org/wiki/Bubblesort>
- <http://siebn.de/index.php?page=anisort/anisort>
- <http://de.wikipedia.org/wiki/Cocktailsort>
- <http://de.wikipedia.org/wiki/Combsort>
- <http://de.wikipedia.org/wiki/Countingsort>
- <http://de.wikipedia.org/wiki/Bucketsort>