

Sortieren- untere Laufzeitschranke und lineare Verfahren

Karsten Klein

Lehrstuhl XI Algorithm Engineering

8. Vorlesung

27.04.06

Überblick

- Eine untere Laufzeitschranke für **allgemeine** Sortierverfahren
- Schnelle „*angepasste*“ Sortierverfahren

*Wie schnell kann ein allgemeines
Sortierverfahren das Sortierproblem im
Worst-Case bestenfalls lösen?*

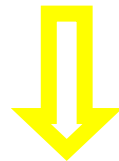
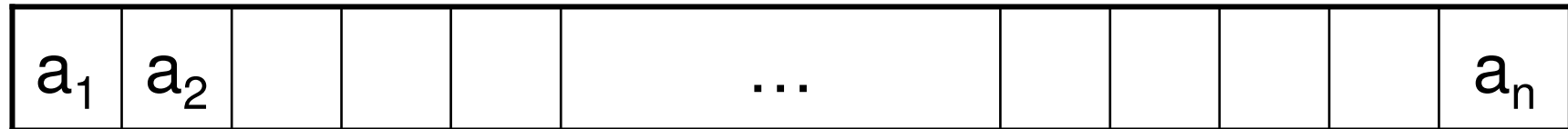
Worst-Case Laufzeiten

- $\Theta(n^2)$: Insertion-, Quick-, Selection-Sort
- $\Theta(n \log n)$: Heap-Sort, Merge-Sort

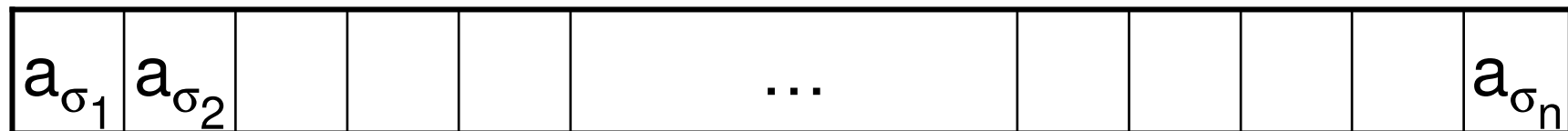
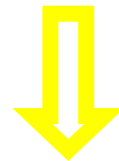
Besser? $O(n)$?

NEIN!

Sortierproblem



Permutation der Eingabeelemente



$\leq \leq$

\dots

\leq

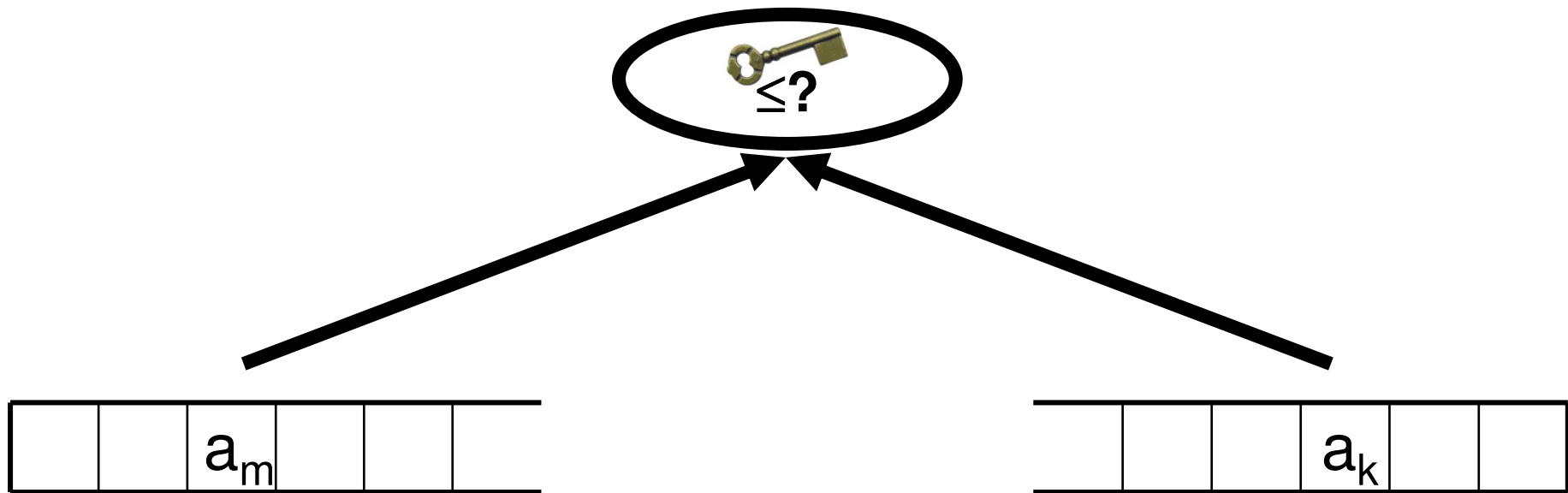
$n!$ Permutationen

Allgemeine Sortierverfahren

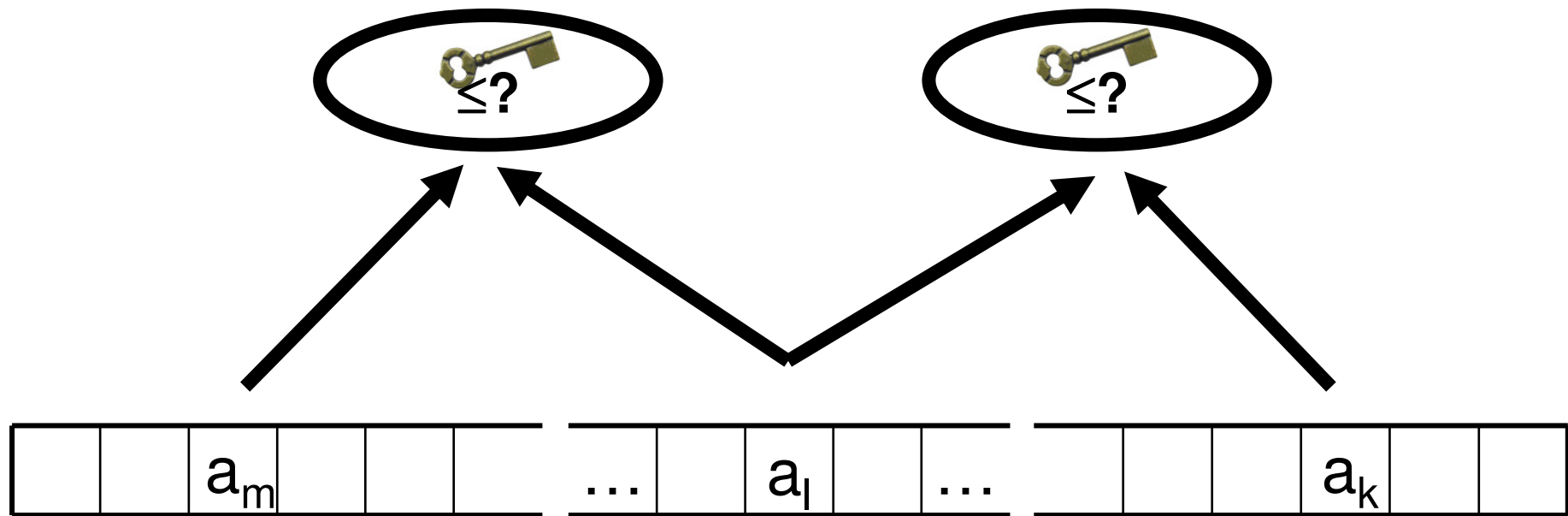
- Vergleich zweier Elementschlüssel
- Vertauschen zweier Elemente

KEIN WISSEN ÜBER SCHLÜSSEL!

Allgemeine Sortierverfahren

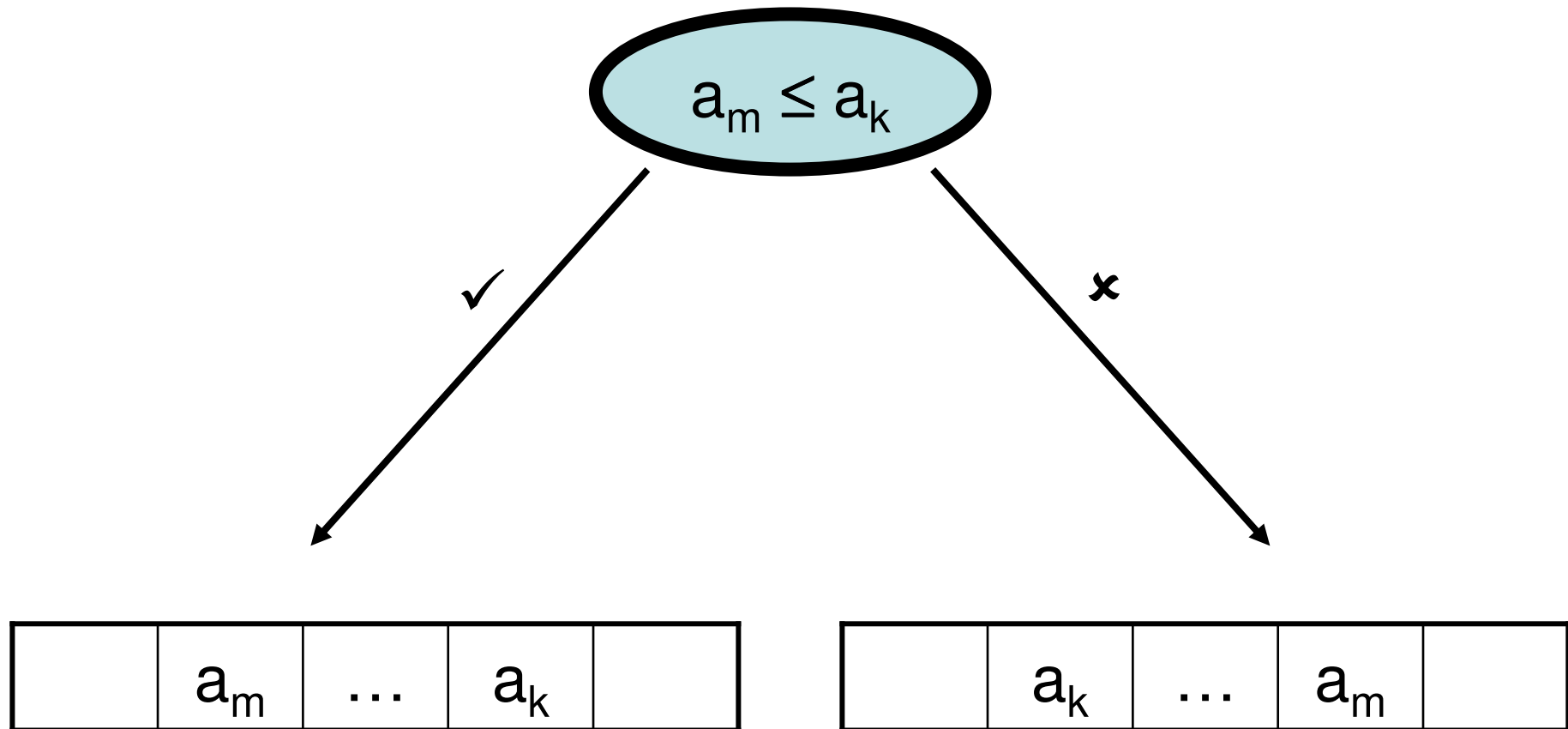


Allgemeine Sortierverfahren

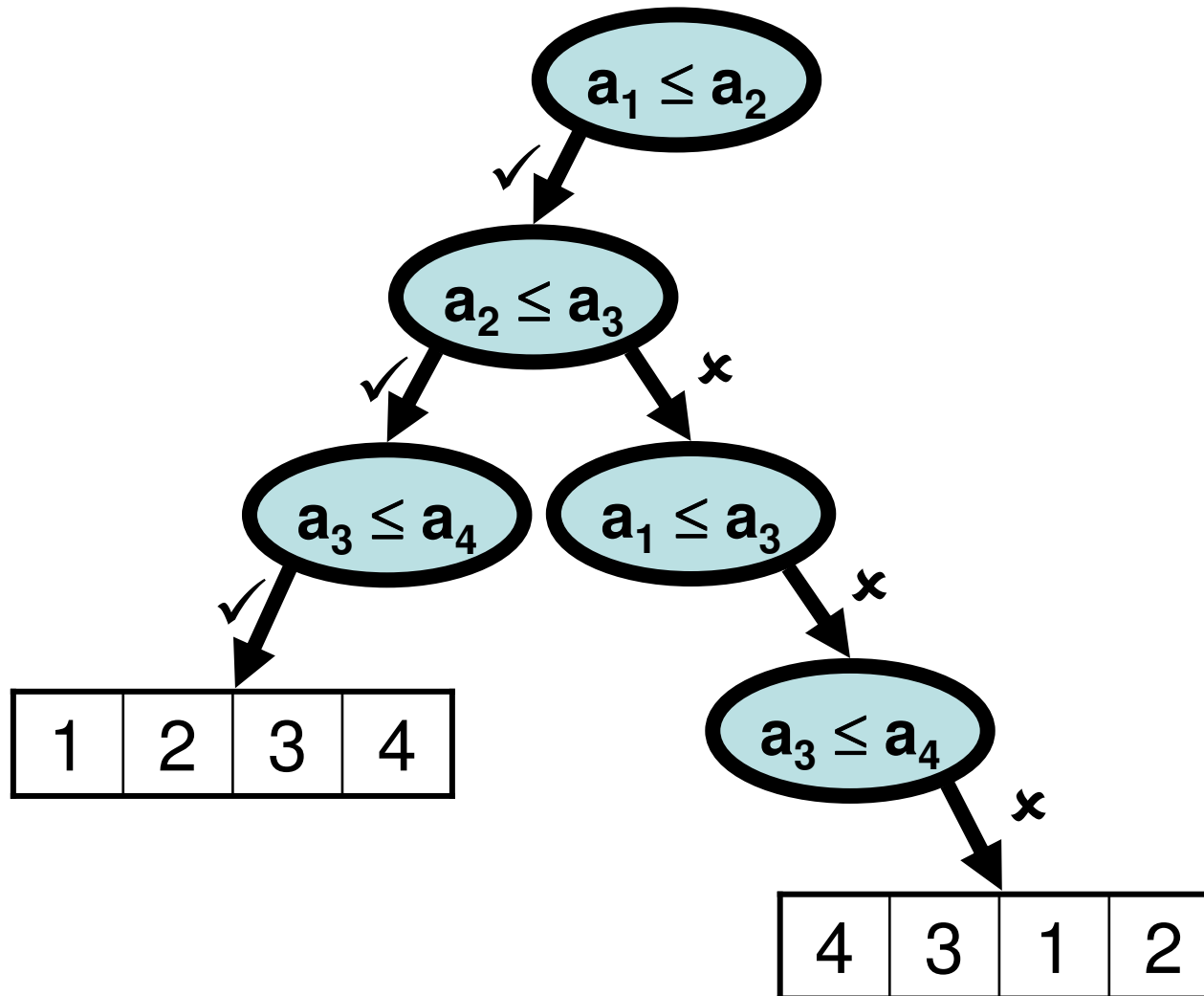


→ Beziehung transitiv ableitbar

Vergleichsentscheidung

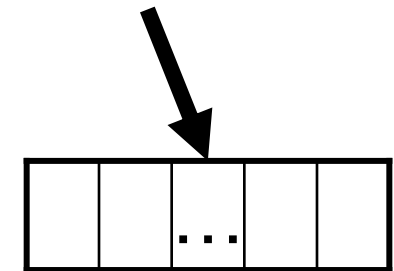
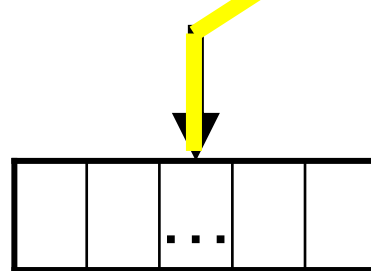
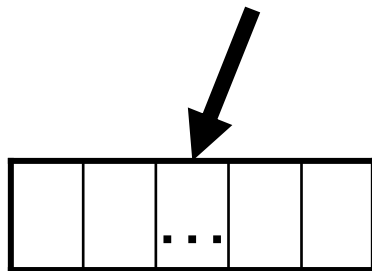
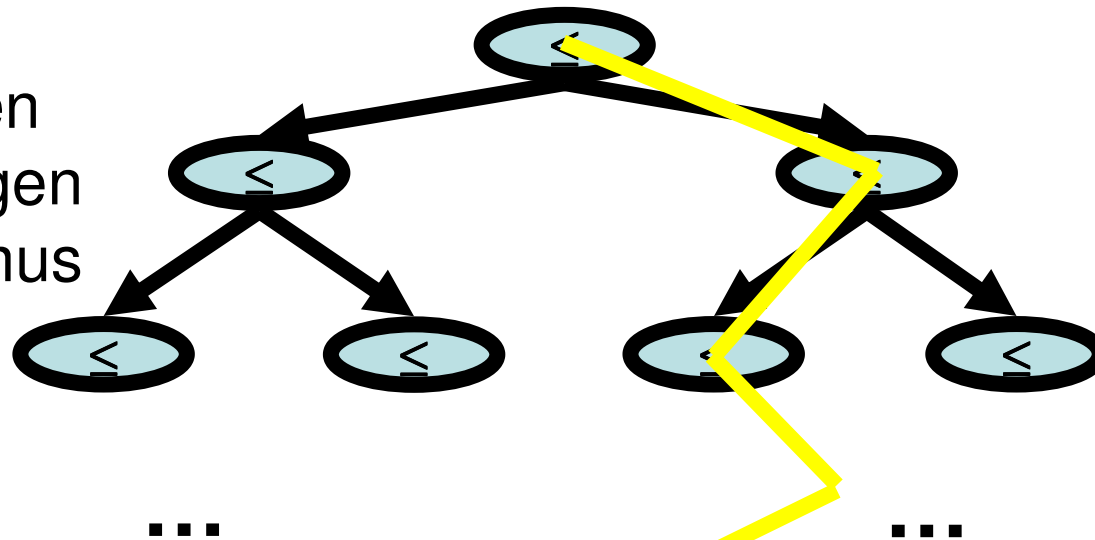


Entscheidungsfolge



Entscheidungsbaum

Alle möglichen
Entscheidungen
des Algorithmus



Alle Permutationen

Entscheidungsbaum

- Knoten entspricht Vergleichsoperation
- Blatt ist Permutation der Eingabe
- Jede Permutation ist als Ergebnis möglich

⇒ $n!$ Blätter in Binärbaum

Worst-Case Schlüsselvergleiche =
Längster Weg von Wurzel zu Blatt

Entscheidungsbaum

Untere Schranke Laufzeit \triangleq
untere Schranke für Baumtiefe t

Binärbaum der Tiefe t hat max. 2^t Blätter \Rightarrow

$$2^t \geq n!$$

$$\Leftrightarrow t \geq \log n!$$

$$\Rightarrow t \geq n/2 (\log n/2) = \Omega(n \log n)$$

Untere Laufzeitschranke

Jedes allgemeine Sortierverfahren benötigt mindestens Worst-Case Laufzeit $\Omega(n \log n)$

„Informationstheoretische untere Schranke“

Heap-Sort und Merge-Sort sind asymptotisch optimal

Lineare Sortierverfahren

Lineare Sortierverfahren

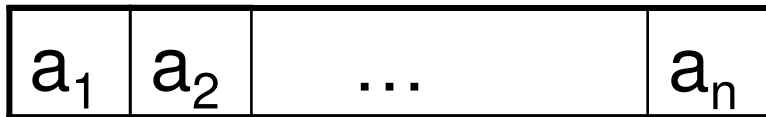
Annahmen über Schlüsselmenge !

- Festes Alphabet (endlich)
- Folgen von Ziffern, Zeichen
- Ganze Zahlen
- Datum (Tag, Monat, Jahr)
- ...

⇒ Nutze arithmetische Eigenschaften der Schlüssel, keine Vergleiche!

Bucket-Sort

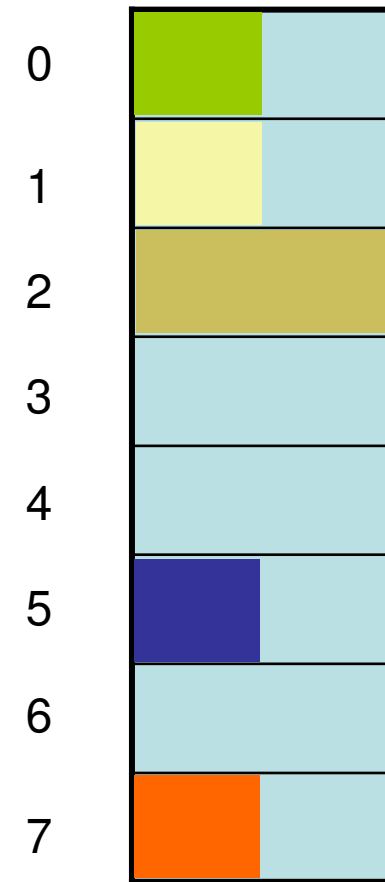
(Sortieren durch Fachverteilung)



Schlüssel aus festem Alphabet
der Größe k
Schlüssel als Index



Verteilungsphase
Sammelphase



Schlüsselwerte

Bucket-Sort

- Buckets für mögliche Schlüsselwerte
- Schlüssel als Index
- Phase I: Verteilungsphase
 - Schlüssel in entsprechendes Bucket hängen
 - Buckets als Liste/Queue
- Phase II: Sammelphase
 - Durchlauf der Buckets nach aufst. Index
 - Hintereinanderhängen der Bucketlisten

Pseudo-Code

```
procedure BUCKETSORT(ref A) {  
  //Verteilung  
  ♦ Initialisiere Bucketfeld  $B$  //Größe  $k$   
  ♦ for  $i := 1, \dots, n$  do {  
     $B[A[i]].put(A[i])$  //Element in Bucketqueue legen  
  }  
  //Sammeln  
  ♦  $i := 1$   
  ♦ for  $j := 0, \dots, k-1$  do { //Jedes Bucket durchlaufen  
    while not  $B[j].isEmpty$  do {  
       $A[i] := B[j].get()$   
       $i := i+1$   
    }  
  }  
}
```

Bucket-Sort

7	4	5	7'	3	1
---	---	---	----	---	---

Verteilungsphase

Sammelphase

1	3	4	5	7	7'
---	---	---	---	---	----

0	
1	1
2	
3	3
4	4
5	5
6	
7	7 7'

Bucket-Sort

Variante für gleichverteilte Zahlen aus $[0, 1)$:

- n gleich große Buckets
- Lege $A[i]$ in $B[\lfloor nA[i] \rfloor]$
- Ungleiche Schlüssel in Bucket \Rightarrow Insertion-Sort, linear

Bucket-Sort

- Sehr einfach
- Auch für reelle Zahlen
- Anpassung an Schlüssel (Indexmapping)
- Bucket-Feld über Wertebereich \Rightarrow
Nur sinnvoll falls nicht zu groß
- Stabil
- Laufzeit $\Theta(n+k)$

Counting-Sort

(Sortieren durch Abzählen, H. Seward 1954)

- Schlüssel sind ganze Zahlen aus $[0..k-1]$
- Nutze Werteanzahl als Index

5, 4, 1, 7, 6, 8, 3

5, 4, 1, 7, 6, 8, 3, 6, 6

Counting-Sort

Element mit Schlüssel i steht rechts von allen Elementen mit Schlüssel $< i$

Schema:

- Zähle Vorkommen von Zahlen (Schlüssel)
- Summiere für Zahl i alle Vorkommen von Zahlen $\leq i$
- Platziere Element mit Schlüssel i entsprechend

Pseudo-Code

procedure COUNTINGSORT(ref A) {

◆ Initialisiere Zählerfeld C mit 0

//Größe k

◆ for $i := 1$ to n do $C[A[i]] := C[A[i]] + 1$

//Zähle an Index

◆ for $i := 1$ to $k-1$ do $C[i] := C[i] + C[i-1]$

//Summiere auf

◆ for $i := n$ to 1 do {

$B[C[A[i]]] := A[i]$

//Eintragen in B

$C[A[i]] := C[A[i]] - 1$

//Runterzählen

}

Kopiere B nach A

}

Beispiel Counting-Sort

$n = 10, k = 12$

A

11	3	2	7	9	4	5	7'	8	1
----	---	---	---	---	---	---	----	---	---

C

0	1	2	3	4	5	6	7	8	9	10	11
	1	1	1	1	1		2	1	1		1

Beispiel Counting-Sort

$n = 10, k = 12$

A

11	3	2	7	9	4	5	7'	8	1
----	---	---	---	---	---	---	----	---	---

C

0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	5	7	8	9	9	10

Beispiel Counting-Sort

$n = 10, k = 12$

A

11	3	2	7	9	4	5	7'	8	1
----	---	---	---	---	---	---	----	---	---



C

0	1	2	3	4	5	6	7	8	9	10	11
	0	1	2	3	4		5	7	8		9

B

1	2	3	4	5	7	7'	8	9	11
---	---	---	---	---	---	----	---	---	----

Counting-Sort

- Sehr einfach
- Schlüssel sind Zahlen
- Schlüssel und Anzahlen als Indizes
- Zählerfeld der Größe $k \Rightarrow$
Nur sinnvoll falls nicht zu groß, z.B.
Graphenalg.
- Stabil
- Laufzeit $\Theta(n+k)$

Radix-Sort

- Schlüssel: Zahlen aus $[0..b^d-1]$,
Ziffernfolge zu einer Basis b
- Sortiere einzelne Ziffern der Schlüssel

Bsp. Postleitzahlen:

44141

53123

66125

Radix-Sort

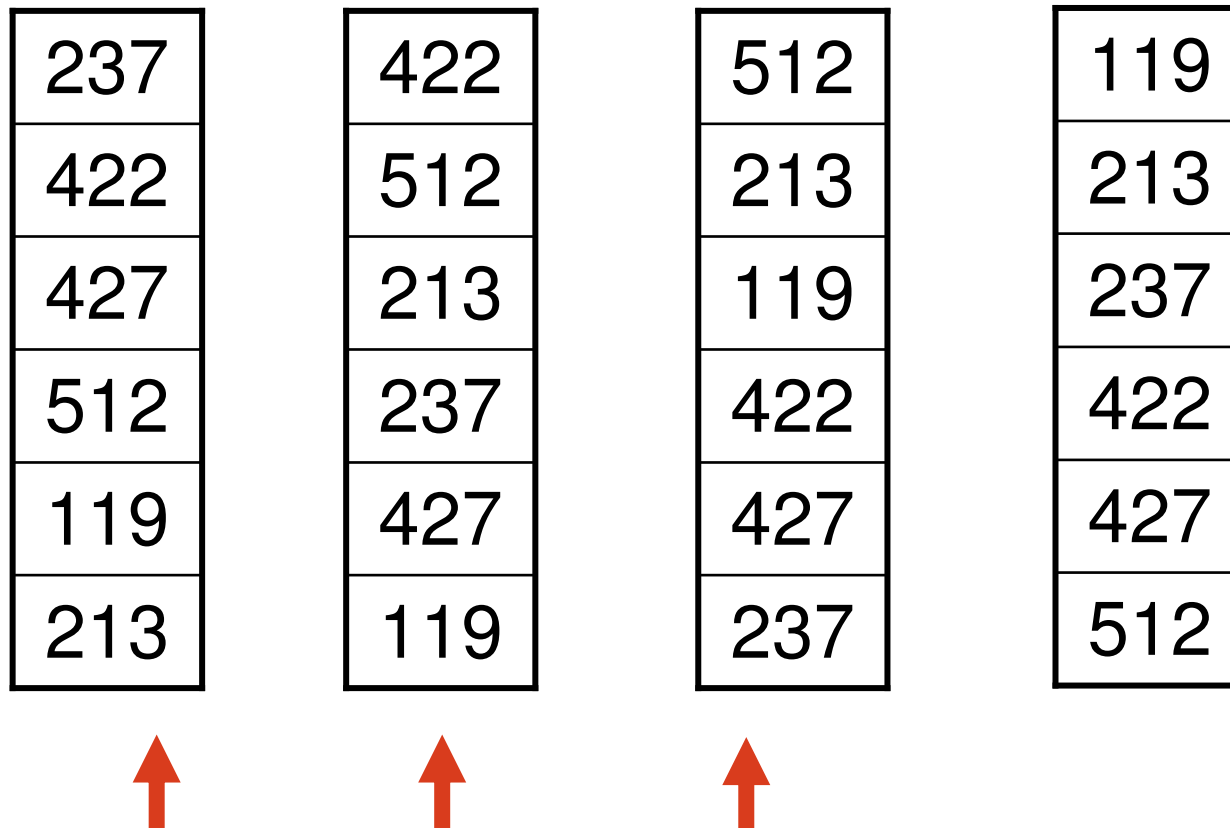
Schema:

- Schlüssel Ziffernfolge
- Sortiere Ziffern einzeln und stabil von hinten nach vorne
- Nach wichtigster=erster Ziffer wird zuletzt sortiert
- Stabilität garantiert Erhalt der Vorsortierung nach i Ziffern

Pseudo-Code

```
procedure RADIXSORT(ref A, int digits) {  
  for  $i := 0$  to  $digits-1$  do  
    Benutze stabiles Sortierverfahren um A nach Stelle  $i$  zu sortieren  
}
```

Beispiel Radix-Sort



Radix-Sort

- Laufzeit abhängig von stab. Sortierverf.
- Mit Counting-Sort: $\Theta(d(n+b))$
- Für d konstant, $b = O(n)$: Linear in n

- Basis Zifferextraktion, $b = 2^m \Rightarrow$
Bitmanipulation abhängig von Software-
/Hardware-Unterstützung
- Nicht für kleine Eingabefolgen
- $b \ll n$

Varianten

- MSD: Most Significant Digits zuerst
 - spare Laufzeit falls Daten schon nach wenigen Ziffern vollständig sortiert
 - „Mehrwege-Quick-Sort“
- LSD: Least Significant Digits zuerst, aber nur auf z.B. erster Hälfte der Ziffern, Rest Insertion-Sort