Kap. 3ff: Untere Laufzeitschranke und Lineare Verfahren

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

Fakultät für Informatik, TU Dortmund

8. VO DAP2 SS 2009 12. Mai 2009

1. Übungstest

- **Termin:** Di 19. Mai 2009, Beginn: 12:15 Uhr (bitte um 12:00 Uhr anwesend sein)
- **Ort:** im Audi-Max (statt Vorlesung)
- **Dauer:** 30 Minuten
- **Stoff:** aus VO-Folien, Skript und Übungen bis Heap-Sort inklusive 3.1.6 Realisierung von Priority Queues durch Heaps

- Ab ca. 12:50 Uhr: Vorlesung bis 13:45 Uhr

Motivation

„Warum soll ich hier bleiben?"

Wir erfahren ein scheinbares Paradoxon

„Was genau ist damit gemeint?"

Gut aufpassen:-)

Überblick

Zuerst:

- Eine untere Laufzeitschranke für ***allgemeine*** Sortierverfahren

Dann:

- Schnelle „*angepasste*" Sortierverfahren, die die obige Schranke „schlagen"

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

a_1	a_2				…					a_n

$\Downarrow$

Permutation der Eingabeelemente

$\Downarrow$

a_{σ_1}	a_{σ_2}				…					a_{σ_n}

$\leq \quad \leq \quad \ldots \quad \leq$

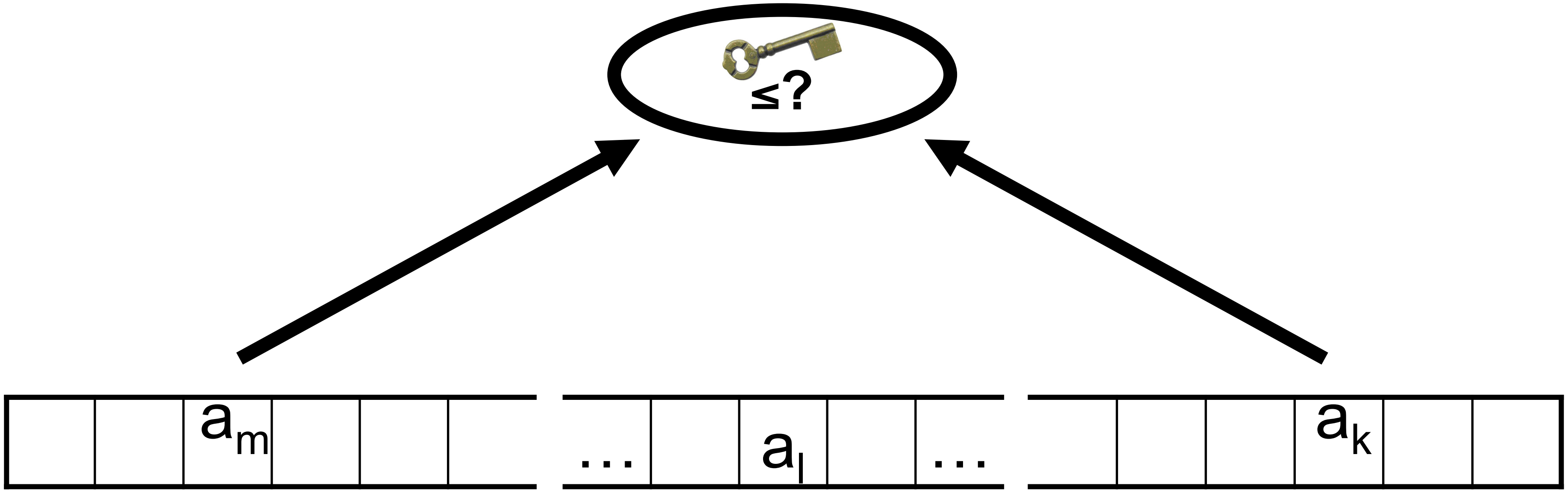
$n!$ Permutationen

Allgemeine Sortierverfahren

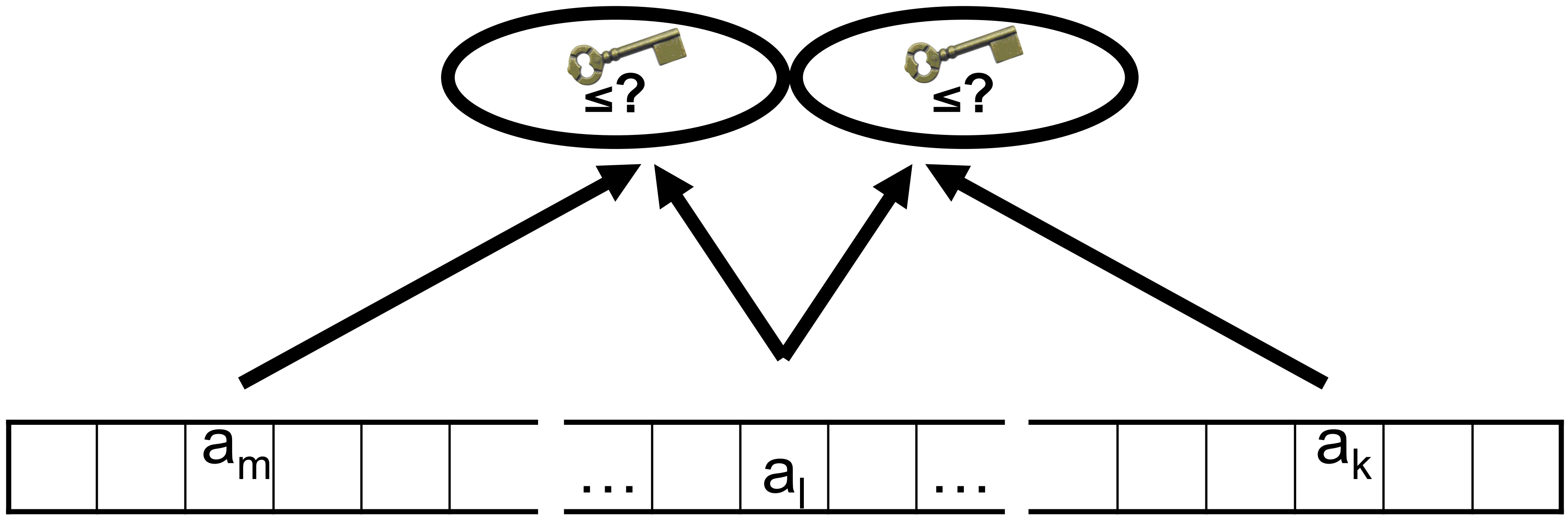
- Vergleich zweier Elementschlüssel
- Vertauschen zweier Elemente

KEIN WISSEN ÜBER SCHLÜSSEL!

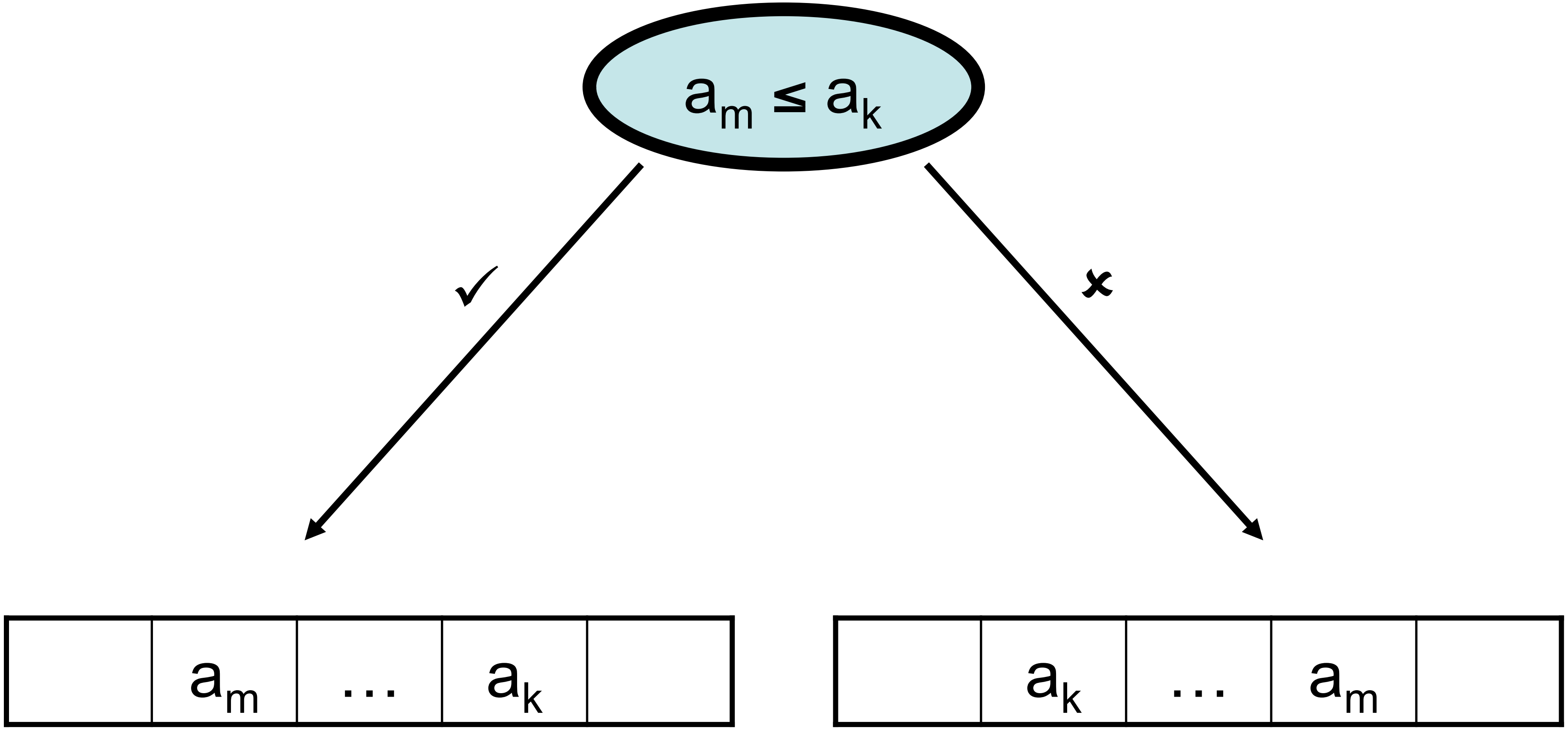
Allgemeine Sortierverfahren



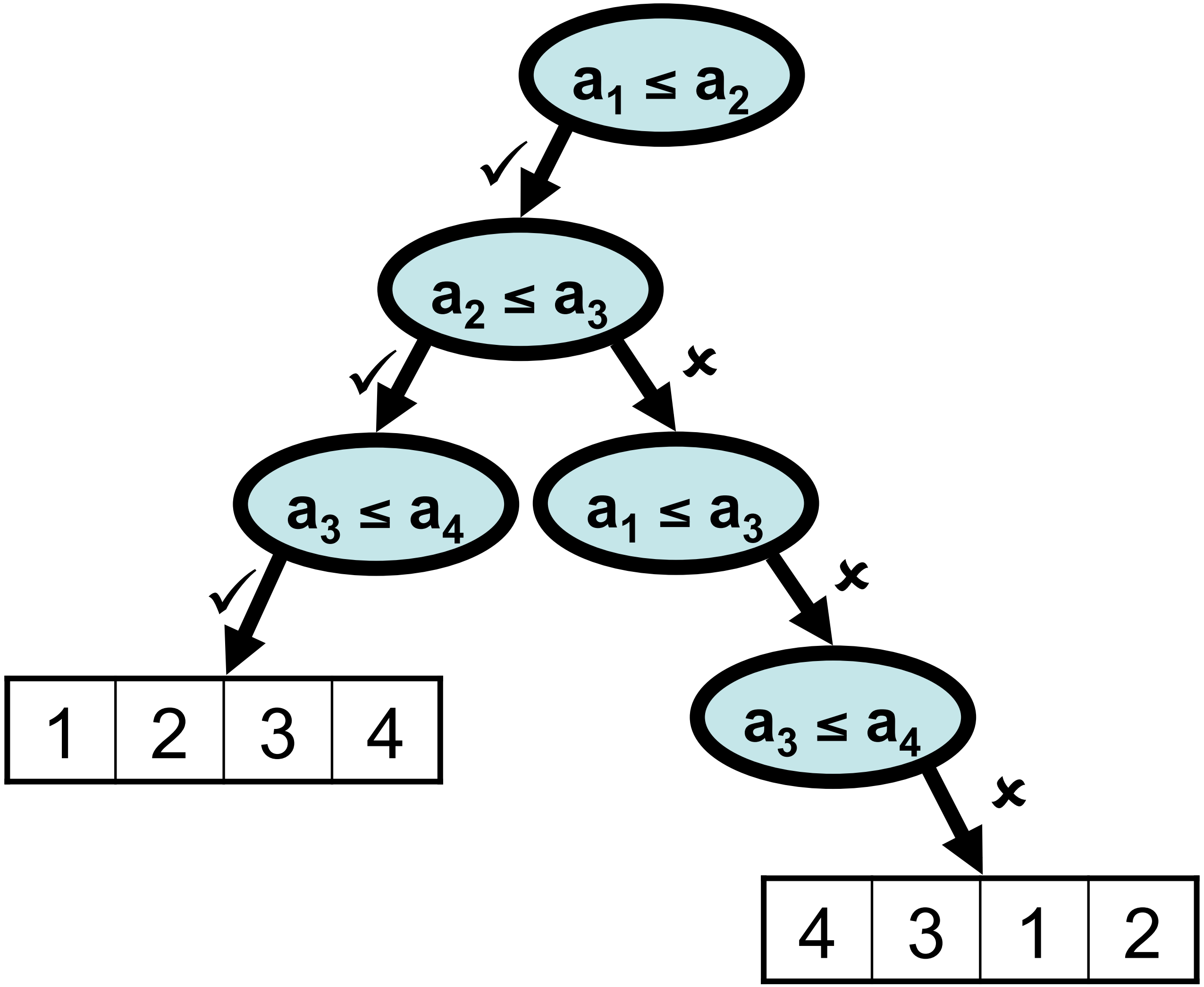
Allgemeine Sortierverfahren



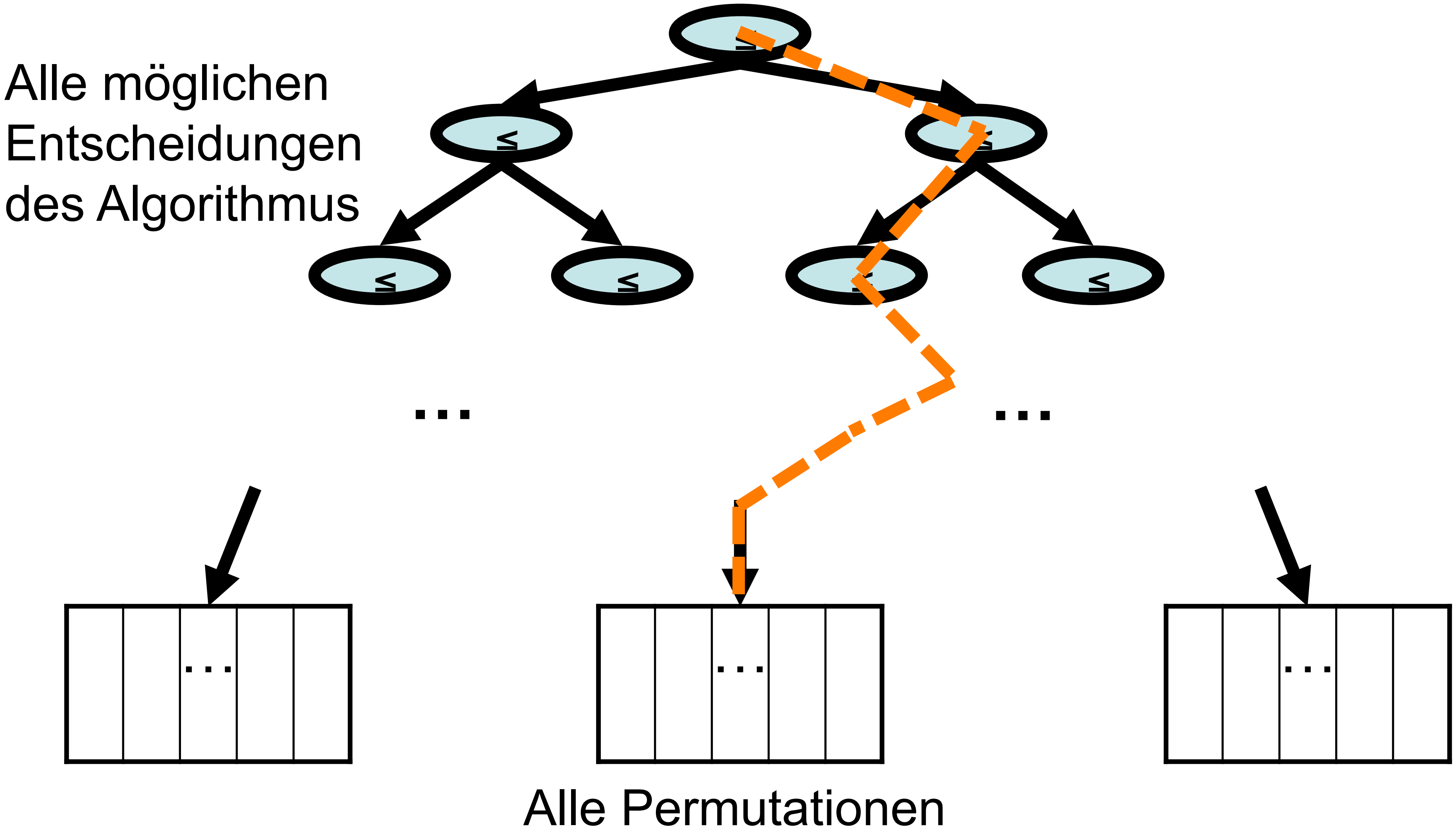
Vergleichsentscheidung



Entscheidungsfolge



Entscheidungsbaum



Entscheidungsbaum

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

$\Rightarrow$ $n!$ Blätter in Binärbaum

Worst-Case Schlüsselvergleiche =
Längster Weg von Wurzel zu Blatt

Entscheidungsbaum

Untere Schranke Laufzeit ≙
untere Schranke für Baumtiefe t

Sei $n \geq 2$. Ein Binärbaum der Tiefe t hat max. 2^t Blätter ⇒

$$2^t \geq n!$$

$$\Leftrightarrow t \geq \log n! = \sum_{i=1}^{n} \log i \geq n/2 \log (n/2)$$

$$\Rightarrow t = \Omega(n \log n)$$

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

Kap. 3.2: Lineare Sortierverfahren

Hier:

- Elemente sind in Feld $A[1..n]$
- Zugriffe auf Schlüssel via $A[i]$

Lineare Sortierverfahren

Annahmen über Schlüsselmenge !

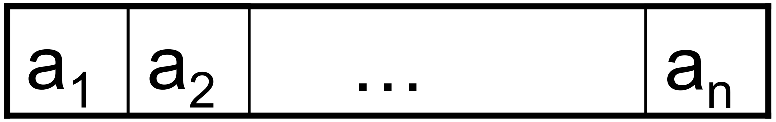
z.B.:
- Festes Alphabet (endlich)
- Folgen von Ziffern, Zeichen
- Ganze Zahlen
- Datum (Tag, Monat, Jahr)
- …

$\Rightarrow$Nutze arithmetische Eigenschaften der Schlüssel, keine Vergleiche!

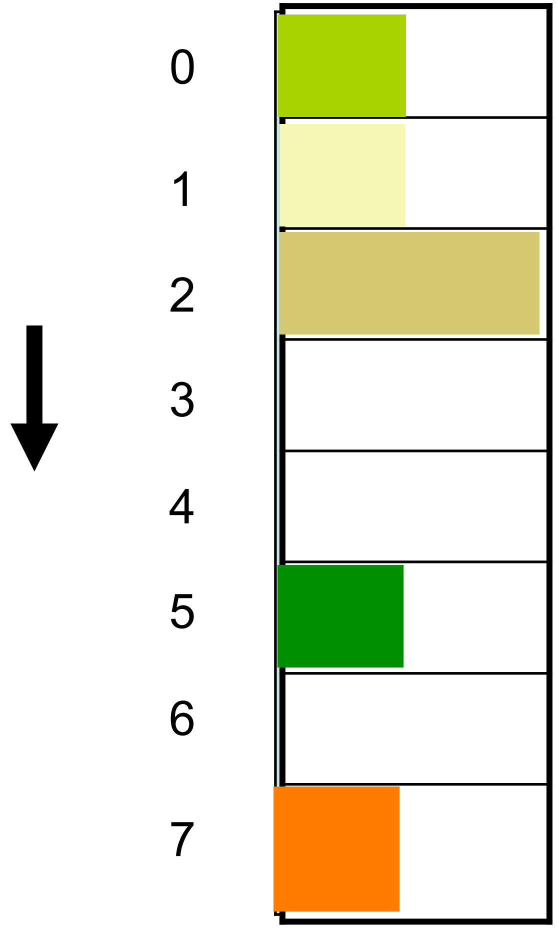
Bucket-Sort

Schlüsselwerte

(Sortieren durch Fachverteilung)



Schlüssel aus festem Alphabet der Größe *k*
Schlüssel als Index



1. Verteilungsphase
2. Sammelphase

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 aufsteigendem Index
  - Hintereinanderhängen der Bucketlisten

Pseudo-Code von BucketSort

```
procedure BUCKETSORT(ref A) {
Initialisiere Bucketfeld B   // Größe k
for i := 1, .., n do {             // Verteilung
        B[A[i]].put(A[i])   // Element in Bucketqueue
    }
i := 1                              // Sammeln
for j := 0, .., 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

0	
1	1
2	
3	3
4	4
5	5
6	
7	7 7‘

Sammelphase

1	3	4	5	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, erwartete Laufzeit: linear, weil nur weniger Zahlen pro Bucket (wegen Gleichverteilung)

Bucket-Sort

- sehr einfach
- auch für reelle Zahlen
- Anpassung an Schlüssel (Indexmapping)
- Bucket-Feld über Wertebereich $\Rightarrow$ nur sinnvoll, falls dies nicht zu groß ist
- 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) {
(1) Initialisiere Zählfeld C mit 0                 // Größe k
(2) for i := 1 to n do C[A[i]] := C[A[i]]+1 //Zähle an Index
(3) for i := 1 to k-1 do C[i] := C[i]+C[i-1] //Summiere auf
(4) for i := n to 1 do {
(5)         B[C[A[i]]] := A[i]                 // Eintragen in B
(6)         C[A[i]] := C[A[i]] – 1             // Runterzählen
(7)}
(8) Kopiere B nach A
(9)}
```

Beispiel Counting-Sort

$n = 10,\ k = 12$

A	11	3	2	7	9	4	5	7‘	8	1

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

Beispiel Counting-Sort

$n = 10, k = 12$

A	11	3	2	7	9	4	5	7‘	8	1

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

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

B	1	2	3	4	5	7	7‘	8	9	11

Counting-Sort

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

Radix-Sort

- Schlüssel: Zahlen aus $[0..b^d\text{-}1]$,
  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

237	422	512	119
422	512	213	213
427	213	119	237
512	237	422	422
119	427	427	427
213	119	237	512

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
- $b << n$

Varianten

- geht auch mit Zeichenfolge: Sortierung von Namen
- 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