

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

2

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

3

Worst-Case Laufzeiten

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

Besser? $O(n)$?

NEIN!

4

Sortierproblem

a_1 a_2 ... a_n

Permutation der Eingabeelemente

a_{σ_1} a_{σ_2} ... a_{σ_n}
 $\leq \leq$... \leq

$n!$ Permutationen

5

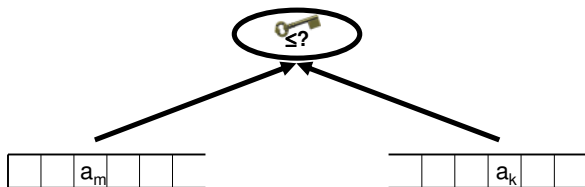
Allgemeine Sortierverfahren

- Vergleich zweier Elementschlüssel
- Vertauschen zweier Elemente

KEIN WISSEN ÜBER SCHLÜSSEL!

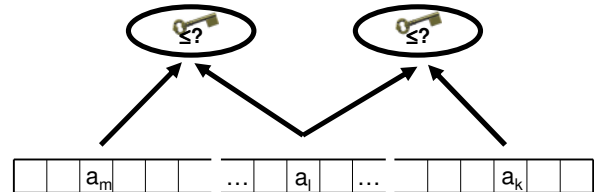
6

Allgemeine Sortierverfahren



7

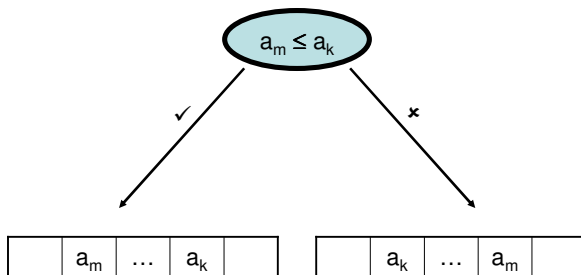
Allgemeine Sortierverfahren



→ Beziehung transitiv ableitbar

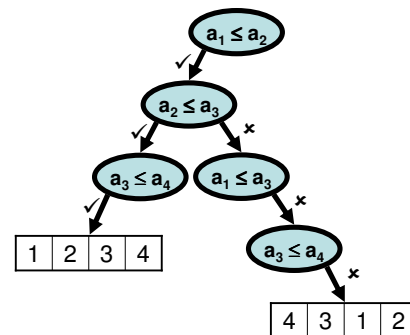
8

Vergleichsentscheidung



9

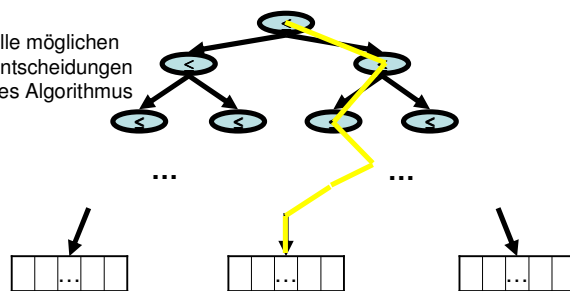
Entscheidungsfolge



10

Entscheidungsbaum

Alle möglichen
Entscheidungen
des Algorithmus



Alle Permutationen

11

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

12

Entscheidungsbaum

Untere Schranke Laufzeit \triangleq
 untere Schranke für Baumtiefe t

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

$$\begin{aligned} 2^t &\geq n! \\ \Leftrightarrow t &\geq \log n! \\ \Rightarrow t &\geq n/2 (\log n/2) = \Omega(n \log n) \end{aligned}$$

13

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

14

Lineare Sortierverfahren

Lineare Sortierverfahren

Annahmen über Schlüsselmenge !

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

\Rightarrow Nutze arithmetische Eigenschaften
 der Schlüssel, keine Vergleiche!

15

16

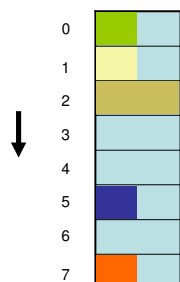
Bucket-Sort

(Sortieren durch Fachverteilung)

$a_1 \ a_2 \ \dots \ a_n$

Schlüssel aus festem Alphabet
 der Größe k
 Schlüssel als Index

Verteilungsphase
 Sammelphase



Schlüsselwerte 17

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

18

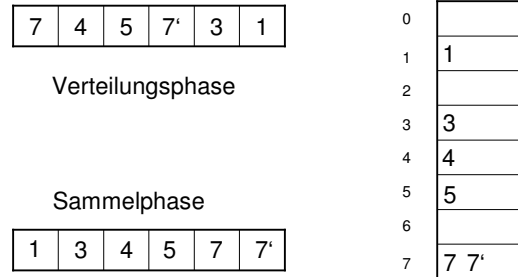
Pseudo-Code

```

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

19

Bucket-Sort



20

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

21

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 $\Theta(n+k)$

22

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

23

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

24

Pseudo-Code

```

procedure COUNTINGSORT(ref A) {
  ♦ Initialisiere Zählfeld 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
}
    
```

25

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

26

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

27

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
---	---	---	---	---	---	---	---	---	----

28

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. Graphenalg.
- Stabil
- Laufzeit $\Theta(n+k)$

29

Radix-Sort

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

Bsp. Postleitzahlen:

44141
53123
66125

30

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

31

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
}
    
```

32

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

↑ ↑ ↑

33

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
- $b \ll n$

34

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

35