

Algorithmen und Datenstrukturen

Wintersemester 2023/24

9. Vorlesung

Sortieren in Linearzeit

Organisatorisches

Diesen Do (16.11.) findet um 8:30 der 1. Zwischentest statt.

Wenn Sie mitschreiben möchten, melden Sie sich in der Abstimmung auf WueCampus bis morgen (15.11.) 15 Uhr an. Wir drucken die Exemplare nach vorhandenen Anmeldungen!

<https://wuecampus.uni-wuerzburg.de/moodle/mod/choice/view.php?id=2802051>

The screenshot shows the WueCampus Moodle interface. The top navigation bar includes the WueCampus logo, 'Dashboard', and 'Meine Kurse'. The left sidebar contains a menu with categories like 'Foren', 'Vorlesung Algorithmen und ...', 'Kursbeschreibung', 'Allgemeine Informationen', 'Vorlesungsfolien', 'Zwischentests', and 'Übungen'. The main content area displays a poll titled 'Teilnahme am 1. Zwischentest' under the heading 'WS22_ADS > Zwischentests > Teilnahme am 1. Zwischentest am 17.11'. The poll options are 'Ja' and 'Nein', and there is a 'Meine Auswahl speichern' button. A red 'X' is overlaid on the poll options, and a red arrow points to it. Another red arrow points to the poll title.

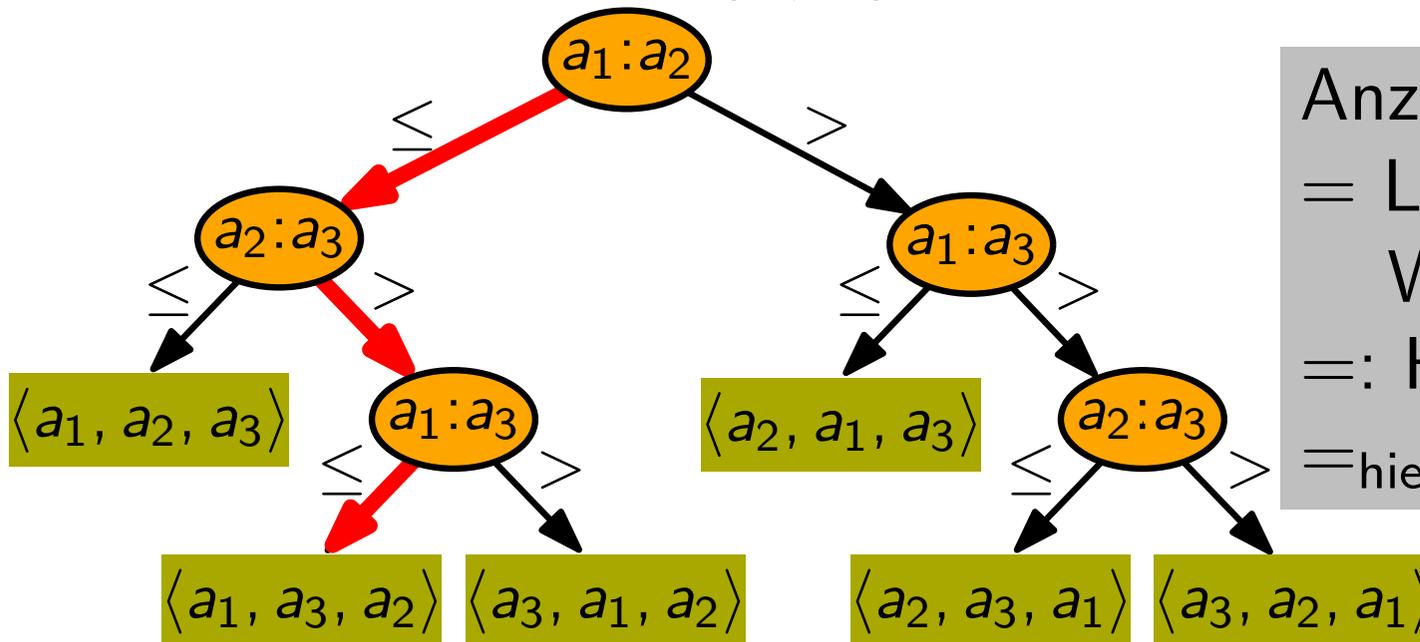
Sortieren durch Vergleichen

Eingabefolge $\langle a_1, a_2, \dots, a_n \rangle$ $\xrightarrow{\text{Sortieralg.}} \text{Ausgabe: sortierte Eingabe}$
Schlüsselvergleiche

Für festes n ist ein *vergleichsbasierter Sortieralg.* charakterisiert durch seinen *Entscheidungsbaum*:



- innere Knoten = Vergleiche (o.B.d.A. links immer \leq , z.B. „ $a_1 \leq a_2$?“)
- Blätter = sortierte Permutationen der Eingabe
- Kanten = Ergebnisse ($\leq / >$) eines Vergleichs



Anz. Vgl. im *worst case*
 = Länge eines *längsten*
 Wurzel-Blatt-Pfads
 =: Höhe des Baums
 = hier 3

Entscheidungsbaum für InsertionSort und $n = 3$ [CLRS]

Eine untere Schranke

Frage: Wie viele Vergleiche braucht *jeder* vergleichsbasierte Sortieralg. im worst case um *n verschiedene* Objekte zu sortieren?

M.a.W. Gegeben

- ein beliebiger vergleichsbasierter Sortieralgorithmus,
- eine Zahl n von verschiedenen Objekten, die man sortieren soll, welche Höhe hat der Entscheidungsbaum *mindestens*?

Beob.: Die Höhe ist eine Funktion der Blätteranzahl.
 Anz. Blätter = Anz. Permutationen von n Obj. = $n!$
 Höhe Binärbaum mit B Blättern $\geq \lceil \log_2 B \rceil$

$$\text{Höhe Entscheidungsbaum} \geq \log_2 n! = \sum_{i=1}^n \log_2 i$$

$$\geq \int_1^n \log_2 x \, dx = \frac{1}{\ln 2} \int_1^n \ln x \, dx = \frac{1}{\ln 2} \int_1^n \mathbf{1} \cdot \ln x \, dx$$

$$= \frac{1}{\ln 2} \left(\mathbf{x} \cdot \ln x \Big|_1^n - \int_1^n \mathbf{x} \cdot \frac{1}{x} \, dx \right) = \frac{(n \ln n - 0) - (n-1)}{\ln 2}$$

$$\in \Omega(n \log n)$$

partielle
Integration

$$\int u' v = uv - \int uv'$$

Resultat

Satz. Jeder vergleichsbasierte Sortieralg. benötigt im schlechtesten Fall $\Omega(n \log n)$ Vergleiche um n Objekte zu sortieren.

Korollar. MergeSort und HeapSort sind *asymptotisch worst-case optimale* vergleichsbasierte Sortieralg.

Wir durchbrechen die Schallmauer

- (● SpaghettiSort sortiert Spaghetti nach Länge ;-)
- CountingSort sortiert Zahlen in $\{0, \dots, k\}$
- RadixSort sortiert s -stellige b -adische Zahlen
- BucketSort sortiert gleichverteilte zufällige Zahlen



aus: www.marions-kochbuch.de



By Eduard Marmet, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=5810282>

CountingSort

- Idee:**
- 1) für jedes x in der Eingabe: zähle die Anzahl der Zahlen $\leq x$
 - 2) benütze diese Information um x im Ausgabefeld direkt an die richtige Position zu schreiben

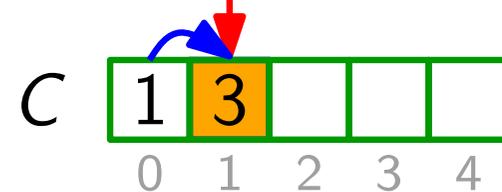
Variable:

| | |
|-----------------|---|
| A Eingabefeld | C Rechenfeld |
| B Ausgabefeld | k begrenzt das <i>Universum</i> : $\{0, \dots, k\}$ |

- Bsp:** 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x



- 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$



CountingSort

- Idee:**
- 1) für jedes x in der Eingabe: zähle die Anzahl der Zahlen $\leq x$
 - 2) benütze diese Information um x im Ausgabefeld direkt an die richtige Position zu schreiben

Variable:

| | | | | |
|-----|-------------|--|-----|---|
| A | Eingabefeld | | C | Rechenfeld |
| B | Ausgabefeld | | k | begrenzt das <i>Universum</i> : $\{0, \dots, k\}$ |

- Bsp:** 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x

| | | | | | | | | | | | | | | | |
|-----|---|---|---|---|---|---|---|---|---------------|-----|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 1 | 2 | 3 | 4 | |
| A | 3 | 0 | 4 | 1 | 3 | 4 | 1 | 4 | \Rightarrow | C | 1 | 2 | 0 | 2 | 3 |

- 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$

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

(Note: In the original image, the value 8 in the C array is crossed out with a red diagonal line and a red arrow points to it from the right.)

- 2) Schreibe jedes x in A direkt an die richtige Position in B

| | | | | | | | | |
|-----|---|---|---|---|---|---|---|---|
| B | | | | | | | | 4 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

(Note: In the original image, a red arrow points from the '4' in the A array to the '4' in the B array, and the '8' in the B array is circled in blue.)

CountingSort

- Idee:**
- 1) für jedes x in der Eingabe: zähle die Anzahl der Zahlen $\leq x$
 - 2) benütze diese Information um x im Ausgabefeld direkt an die richtige Position zu schreiben

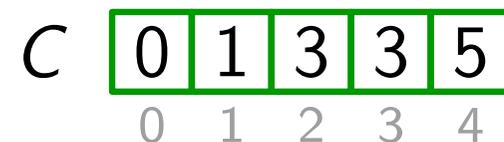
Variable:

| | | | | |
|-----|-------------|--|-----|---|
| A | Eingabefeld | | C | Rechenfeld |
| B | Ausgabefeld | | k | begrenzt das <i>Universum</i> : $\{0, \dots, k\}$ |

- Bsp:** 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x



- 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$



- 2) Schreibe jedes x in A direkt an die richtige Position in B



CountingSort ist *stabil!*

CountingSort

- Plan:**
- 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x
 - 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$
 - 2) Schreibe jedes x in A direkt an die richtige Position in B

CountingSort(int[] *A*, int[] *B*, int *k*)

sei $C[0..k] = \langle 0, 0, \dots, 0 \rangle$ ein neues Feld

for $j = 1$ **to** $A.length$ **do** // (1a)

// $C[x]$ enthält jetzt die Anz. der Elem. gleich x in A

for $i = 1$ **to** k **do** // (1b)

// $C[x]$ enthält jetzt die Anz. der Elem. $\leq x$ in A

for $j = A.length$ **downto** 1 **do**

┌ // (2)

Aufgabe:

Fülle die Felder mit Code, der obige Idee umsetzt!

CountingSort

Laufzeit:
 $O(n + k)$

- Plan:**
- 1a) Für jedes x in A , zähle die Anz. der Zahlen gleich x
 - 1b) Für jedes x in A , berechne die Anz. der Zahlen $\leq x$
 - 2) Schreibe jedes x in A direkt an die richtige Position in B

CountingSort(int[] A , int[] B , int k)

Eingabefeld
Ausgabefeld
beschränkt Universum $\{0, \dots, k\}$

sei $C[0..k] = \langle 0, 0, \dots, 0 \rangle$ ein neues Feld

for $j = 1$ **to** $A.length$ **do** $C[A[j]] = C[A[j]] + 1$ // (1a)

// $C[x]$ enthält jetzt die Anz. der Elem. gleich x in A

for $i = 1$ **to** k **do** $C[i] = C[i] + C[i - 1]$ // (1b)

// $C[x]$ enthält jetzt die Anz. der Elem. $\leq x$ in A

for $j = A.length$ **downto** 1 **do**

$\left[\begin{array}{l} B[C[A[j]]] = A[j] \\ C[A[j]] = C[A[j]] - 1 \end{array} \right.$ // (2)

RadixSort

(Jahr, Monat, Tag)



Frage: Gegeben Liste von Menschen mit deren Geburtstagen.
Wie würden Sie die Liste nach Alter sortieren?

Drei (?) Lösungen:

- Geburtstage in Anz. Tage seit 1.1.1970 umrechnen, dann vergleichsbasiertes Sortierverfahren verwenden.
- Spezielle Vergleichsroutine schreiben und in vergleichsbasiertes Sortierverfahren einbauen.
- Liste $3 \times$ sortieren: je $1 \times$ nach Jahr, Monat, Tag.

Aber in welcher Reihenfolge??

RadixSort(A, s)

Anz. Stellen (hier: 3)

Laufzeit?

for $i = 1$ **to** s **do** [1 = Index der *niederwertigsten (!)* Stelle]
 └ sortiere A *stabil* nach der i -ten Stelle

z.B. mit CountingSort

Beispiel

Sortiere $A = \langle 25, 13, 31, 23, 11, 37, 15 \rangle$:

Gemäß RadixSort erst nach Einern, dann (stabil) nach Zehnern.

$A_1 = \langle 31, 11, 13, 23, 25, 15, 37 \rangle$

$A_2 = \langle 11, 13, 15, 23, 25, 31, 37 \rangle$ ✓

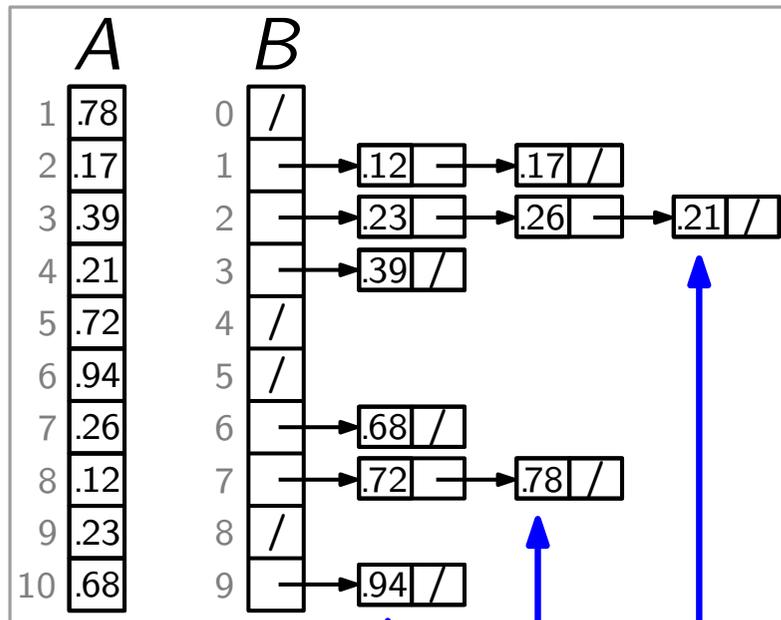
RadixSort(A, s)

for $i = 1$ **to** s **do**

 ⊥ sortiere A *stabil* nach der i -ten Stelle

BucketSort

[CLRS]



(c) www.seafish.org

„Eimerinhalt“: Verkettete Liste von Elementen aus A.

Hilfsfeld $B[0..n - 1]$;

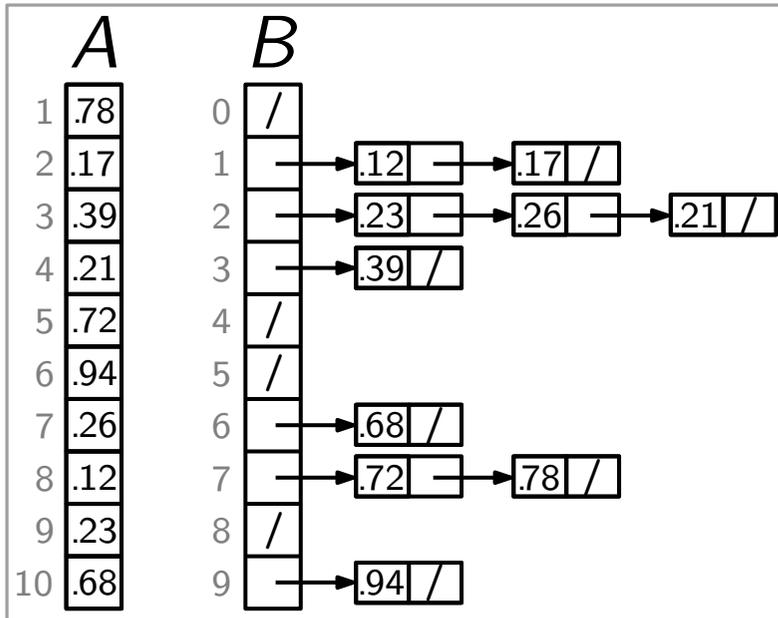
jeder Eintrag entspricht einem „Eimer“ der Weite $1/n$

Eingabefeld $A[1..n]$ enthält Zahlen,
zufällig und gleichverteilt aus $[0, 1)$ gezogen

[Im Bsp. auf 2 Nach-
kommastellen gerundet!]

BucketSort

[CLRS]



BucketSort(Feld A von Zahlen in $[0, 1)$)

$n = A.length$

lege Feld $B[0..n - 1]$ von Listen an

for $j = 1$ **to** n **do**

 füge $A[j]$ in Liste $B[\lfloor n \cdot A[j] \rfloor]$ ein

for $i = 0$ **to** $n - 1$ **do**

 sortiere Liste $B[i] = \left[\frac{i}{n}, \frac{i+1}{n} \right) \cap A$

hänge $B[0], \dots, B[n - 1]$ aneinander
kopiere das Ergebnis nach $A[1..n]$

Korrektheit?

- 2 Fälle:
- $A[i]$ und $A[j]$ in der gleichen Liste
 - $A[i]$ und $A[j]$ in verschiedenen Listen

Laufzeit?

- *erwartet*, hängt von den zufälligen Zahlen in A ab
- hängt vom Sortieralgorithmus in Zeile 6 ab;
wir nehmen InsertionSort: schnell auf kurzen Listen!

Erwartete Laufzeit von BucketSort

$$T_{\text{BS}}(n) = \Theta(n) + \sum_{i=0}^{n-1} T_{\text{IS}}(n_i) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

$$E[T_{\text{BS}}(n)] = E[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) = \Theta(n)$$

Behauptung: $E[n_i^2] \leq 2 - \frac{1}{n}$

Beweis. Def. Indikator-ZV $X_j := 1$, falls $A[j]$ in Eimer i fällt. *fest!*

$$\Rightarrow n_i = \sum_{j=1}^n X_j \quad E[X_j] = \Pr[X_j = 1] = 1/n$$

$$\Rightarrow n_i^2 = \left(\sum_{j=1}^n X_j \right)^2 = \sum_{j=1}^n \sum_{k=1}^n X_j X_k$$

$$= \sum_{j=1}^n X_j^2 + \sum_{j=1}^n \sum_{k \neq j} X_j X_k$$

Erwartete Laufzeit von BucketSort

Behauptung:
 $E[n_i^2] \leq 2 - \frac{1}{n}$

$$\text{Es gilt } n_i^2 = \sum_{j=1}^n X_j^2 + \sum_{j=1}^n \sum_{k \neq j} X_j X_k$$

$$\Rightarrow E[n_i^2] = \sum_{j=1}^n E[X_j^2] + \sum_{j=1}^n \sum_{k \neq j} E[X_j X_k]$$

Behandle die beiden Typen von Erwartungswerten getrennt:

$$\begin{aligned} E[X_j^2] &= 1 \cdot \Pr[X_j^2 = 1] + 0 \cdot \Pr[X_j^2 = 0] && \text{unabhängig von } j! \\ &= 1 \cdot \Pr[X_j = 1] + 0 \cdot \Pr[X_j = 0] = 1 \cdot \frac{1}{n} + 0 = \frac{1}{n} \end{aligned}$$

$$E[X_j X_k] = E[X_j] \cdot E[X_k] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2} \quad \leftarrow \text{unabh. von } j \text{ und } k!$$

für $j \neq k$ sind X_j und X_k unabhängig

Fasse die Zwischenergebnisse zusammen:

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^n E[X_j^2] + \sum_{j=1}^n \sum_{k \neq j} E[X_j X_k] \\ &= n \cdot \frac{1}{n} + n \cdot (n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n} \quad \square \end{aligned}$$

Zusammenfassung

- Jedes *vergleichsbasierte* Sortierverfahren braucht im schlechtesten Fall $\Omega(n \log n)$ Vergleiche für n Zahlen.
- **CountingSort** sortiert Zahlen in $\{0, \dots, k\}$ (*stabil!*)
Laufzeit für n Zahlen: $O(n + k)$
- **RadixSort** sortiert s -stellige b -adische Zahlen
Laufzeit für n Zahlen: $O(s \cdot (n + b))$
- **BucketSort** sortiert gleichverteilte zufällige Zahlen
erwartete Laufzeit für n Zahlen: $O(n)$

Bem. Die Idee mit den (gleichgroßen) Eimern ist natürlich nicht nur auf Zufallszahlen beschränkt, aber hier lässt sie sich hübsch analysieren.